2005:181 CIV

MASTER'S THESIS

Development of a Pharmacy Management Application using the Framework MSTORE

Javier Gonzalez Pisano

Luleå University of Technology

MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of Computer Science

2005:181 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--05/181--SE

Development of a Pharmacy Management Application using the framework MSTORE

Javier González Pisano

Luleå University of Technology

May 2005

Supervisors:
Aquilino Juan De La Fuente (Oviedo University)
Kåre Synnes (Luleå University of Technology)

Abstract

Store management is an application that is complex although it seems apparently simple.

A good store management system should prevent a wide set of issues: some of them inside the own business logic of the application (for instance employees or providers management) and others depending on the application's architecture that makes the system more open.

The goal of this Master Thesis Project is to define a Framework for the design of Store Management Applications, in collaboration with the Master Thesis "Development of a Ironmonger Management Application using the framework MSTORE".

This work is aimed towards SMEs, providing attractive and viable software tools that allow them their modernization for reaching the competitively grade of the department stores (cooperating with other parallel modules in the same research program).

Three levels compose the Framework: in the top there is the interface layer, with all the common behaviour for all the standard stores. In the second layer, it can be found the abstract level that implements a part of the first level operations. Finally in the third level there will have the specific classes for each application that can be done using the Framework.

For testing the Framework usefulness and good operation there were developed two concrete applications, each one with specific problems. The first is a Pharmacy, which works with perishable products, so the software should provide functions for controlling the sell by dates and maintaining the merchandise in good conditions. The second is an Ironmonger, which needs a special support for buying sets and selling units. Both applications have also some common behaviour that is developed in the same way.

This report will describe the concepts of the framework in relation to problems and solutions regarding store management. It also contains the main points about the analysis, implementation and appearance of the Pharmacy application.

Keywords

Framework, Design Patterns, .NET, C#, Store Management, SMEs, Web Service

Abstract in Spanish

Introducción

El proyecto que se ha realizado forma parte de un programa de creación de componentes software desarrollado en la Universidad de Oviedo con el objetivo de facilitar a las PYMES (Pequeñas y Medianas empresas) con herramientas software viables que permitan su modernización, pudiendo así reducir costes y competir con almacenes con mayores recursos.

El programa esta compuesto por varios proyectos: Gestión de Documentos, Gestión de Riesgos o el que se va a explicar en este documento: Gestion de Almacenes.

Dada la extensión del proyecto se decidió a su comienzo dividirlo en dos, delimitando las partes que incluiría cada uno. Los dos proyectos fueron realizados paralelamente y en colaboración, y al termino de su desarrollo se juntaron. Este documento explica como se implementó una de las partes, la otra parte se puede ver en el documento "*Development of a Ironmonger using the framework MSTORE*", realizado por María Rodríguez Fernández.

Conceptos relacionados

La herramienta software que se quiere implementar va a ser definida en forma de framework [6][25]. Un framework se puede definir como una colaboración de clases adaptables que definen una solución para un problema dado. Esta colaboración debería:

- Definir las abstracciones principales y sus interfaces.
- Establecer las relaciones entre los objetos.
- Adaptar a soluciones particulares usando redefinición.
- Incluir alguna solución por defecto.

Los patrones de diseño están muy relacionados con los frameworks orientados a objetos, ya que ambos facilitan la reutilización capturando estrategias exitosas de desarrollo software. La mayor diferencia consiste en que los frameworks se centran en diseños concretos, mientras que los patrones constituyen diseños abstractos. Los patrones dicen *cómo* resolver un problema, mientras que los frameworks proporcionan una *solución* para el problema.

Esta ampliamente aceptada la conveniencia de documentar frameworks usando patrones [3][14]. El mayor propósito del conjunto de patrones es mostrar como usar un framework, no cómo funciona, pero los patrones pueden mostrar una gran parte del diseño. Cada patrón describe un problema común en el dominio del problema, y a continuación se describe como resolverlo. La documentación de un framework tiene tres propósitos, y los patrones pueden satisfacer los tres, ya que describen la intención del framework, cómo usarlo y el diseño detallado de este.

JHotDraw[10] es un buen ejemplo del uso de patrones para describir un framework. Se usa para el desarrollo de aplicaciones gráficas en Java, y fue documentado en términos de patrones. En JHotDraw, los patrones guiaron la estructura del diseño.

Objetivo

Concretando, el objetivo del proyecto es definir un framework para la gestión de almacenes que cualquier PYME pueda usar directamente para solucionar sus problemas concretos. El framework debe ser pues lo bastante genérico como para ser usado en distintos tipos de almacenes, teniendo en cuenta los aspectos más importantes de este tipo de aplicaciones. Se buscará el mayor grado de flexibilidad posible, pero teniendo en cuenta que la captura de todos los aspectos posibles conllevaría un tiempo de desarrollo mucho mayor del estimado para este proyecto. Los aspectos que no se han podido recoger se tendrán en cuenta en futuras versiones del mismo.

El framework va a utilizar un esquema ya usado en otros frameworks como JHotDraw con beneficios probados. El diseño va a ser dividido en 3 capas con varias características:

- La primera capa, basada en interfaces, define el comportamiento básico del framework así como sus interacciones, sin ninguna implementación.
- La segunda capa, de clases abstractas, incluye la implementación de cualquier comportamiento que se considere común a todos los programas de gestión de almacenes, como por ejemplo el manejo básico del stock de los artículos o la distribución en planta del almacén.
- El tercer nivel estará formado por la implementación concreta de la solución de un cliente con sus características particulares: en el ejemplo el cliente podria usar la distribución en planta que incluye el segundo nivel o modificarla de acuerdo a sus necesidades.

Como el framework trata de ser lo más genérico posible, la abstracción de las entidades debería ser lo más independiente posible de cualquier detalle de implementación. La separación en los dos primeros niveles (interfaces y clases abstractas) facilita la diferenciación del diseño del framework y su implementación. El tercer nivel también tiene valor, ya que además de demostrar que el Framework funciona va a proporcionar algunos ejemplos al programador que quiera usarlo por primera vez.

Requisitos

Un vistazo a algunos de los programas más usados de gestión de almacenes [2][7] y en bibliografía básica de gestión de almacenes [1][2][5] muestran los requisitos generales que debería tener la librería. Esta es solo una aproximación general, se han elaborado listas de requisitos mas detalladas que no se incluyen en esta documentación:

- **Artículos:**
    - Gestión del stock automática, gestión de códigos, modelado de artículos perecederos (con fecha de caducidad y sin fecha de caducidad), artículos gestionados por lotes, rangos, familias, catálogos, gestión del precio y beneficios, control de unidades deterioradas, productos sustitutivos.
- **Entradas y salidas de productos**:
    - Gestión de todo lo relativo a **entradas** en el almacén: zonas de entrega, productos contrastados con albaranes o distribucion en las estanterías con indicación de las rutas a usar.

    - Gestión de las **salidas** en el almacén: preparación de pedidos, empaquetado, recogida de productos minimizando los movimientos a las estanterías, etc.

- **Gestión de pedidos:**
    - Gestión del stock basado en los métodos tradicionales (stock mínimo) o en las políticas clásicas de gestión de stocks. Compras a los proveedores, ventas a los clientes y gestión de los distintos tipos de documentos (presupuestos, albaranes, tickets, facturas...).
- **Movimientos**:
    - Se permiten cambios de mercancía entre zonas y también entre almacenes. Almacenamiento de productos en la tienda. Es importante saber en todo momento el estado y la localización de todos los productos en la compañía, así como realizar inventarios.
- **Gestión de recursos:**
    - Empleados, maquinaria, control de costes...
- **Almacén** (la implementación debe ser multialmacén):
    - **Distribución en planta:** Proporciona distintos tipos de equipamiento como por ejemplo pasillos.
    - **Distribución de artículos**: Es necesario tener mecanismos que localicen espacios libres para las entradas de productos, así como realizar el proceso simétrico para las salidas. Estos métodos podrán ser distintos dependiendo de la zona del almacén en la que nos encontremos. Por ejemplo, es útil usar un método FIFO cuando trabajamos con productos perecederos en la zona.
    - **Distribución física:** Métodos para hacer una representación gráfica del almacén.
- **Servicios Web**: Las principales operaciones deberían tener un interfaz como servicio web, de manera que distintas aplicaciones puedan acceder a ellas.
- **Gestión de la base de datos**: Capa objeto-relacional que encapsule el acceso a la base de datos.

De estos requisitos se puede deducir una estructura general de paquetes, que fueron repartidos entre los dos proyectos para su implementación y prueba, y que posteriormente fueron juntados para crear el framework.

La versión final contiene más de 200 archivos de código (contando interfaces y clases abstractas) distribuidas en 16 paquetes, aparte de un número similar para cada una de las aplicaciones de ejemplo que se construyeron usando el framework y se explicarán posteriormente.

Diseño del framework

Como ya se indicó, los patrones pueden ayudar en gran medida a resolver los problemas encontrados a lo largo del diseño. Un framework puede ser entendido como una colección de familias de patrones unidos para resolver un dominio de aplicación concreto. Los patrones pueden ser vistos como los elementos que componen la arquitectura del framework.

En la documentación del framework MSTORE se explican algunos de los problemas encontrados a lo largo del diseño y la solución dada, expresada como un patrón si esa ha sido la elección, y si no lo es motivando las razones por las que se ha escogido otra solución. Antes de elegir la solución, se valoran otras posibles soluciones y se explica por que la elegida es la más apropiada. Si la solución es obvia, solo se explican los beneficios de ésta.

Hay varias conclusiones que se pueden tomar del uso de patrones durante el diseño:

- El proceso de desarrollo de una librería tan amplia nos muestra que en muchas ocasiones la mejor característica de un buen diseño es su simplicidad, luego en muchas ocasiones los patrones no son la mejor solución ya que complican el diseño en exceso, abogando por soluciones más sencillas aunque menos flexibles.
- En otros casos ningún patrón se adapta a la estructura del problema que buscamos: en ocasiones se puede modificar la estructura del patron para adaptarlo al problema, pero la solución mas preferable sera generalmente no usar ningún patron y realizar un diseño particular para dicho problema.
- A veces resulta difícil decidirse entre varios patrones: generalmente habrá que evaluar los pros y contras que ofrecen cada uno de ellos y decidirse por uno, dejando de lado ventajas que ofrecen otros. En otras ocasiones también podría convenir usar una solución mixta, pero por lo general ésta es demasiado complicada.

En la documentación definitiva fueron incluidos los problemas más relevantes de los encontrados a lo largo del diseño, elegidos debido a su validez académica o porque representan aspectos importantes de la estructura del framework.

Una vez que se resolvieron todos los problemas particulares, el siguiente paso consistió en ponerlos todos juntos para obtener el diseño del framework, y posteriormente los dos subproyectos se juntaron para hacer la versión definitiva.

Como puede deducirse ésta no es una tarea lineal y fué hecha en varias iteraciones, pero se espera que de esta manera el producto logrado sea consistente. El framework completo puede ser encontrado en la página web del proyecto [12]. Sin embargo éste aún se encuentra en versión alfa, necesitando que futuros proyectos continúen con su desarrollo, probándolo y ampliándolo.

Desarrollo una farmacia usando MSTORE

Tras el diseño de la librería, se han desarrollado un par de aplicaciones usándola, demostrando así que funciona con distintos tipos de almacenes y proporcionando además ejemplos al programador que quiera usarla por primera vez. En este caso la aplicación se trata de un programa de gestión del almacén de una farmacia, con las problemáticas particulares que éste puede tener. El otro proyecto paralelo se centró en la implementación de una aplicación de gestión de un almacén para una ferretería. Se intentó que ambas aplicaciones usaran partes distintas del framework, de manera que entre ambas le dieran la mayor cobertura posible de cara a probarlo. El resultado es que se ha probado que funciona gran parte, aunque aún hay partes pendientes de implementar o probar.

A la hora de diseñar el funcionamiento de la farmacia, se observó que algunas de sus funciones podían ser completadas con el framework sin necesidad de ninguna modificación en éste, mientras que otras necesitaban modificar el diseño existente o añadir características a las ya existentes. Una vez que se modificaron /ampliaron dichas características, muchas de ellas pasaron a formar parte de una segunda versión del framework más completa. Ésta es otra ventaja del desarrollo de aplicaciones usando el framework, según se implementan más éstas van añadiendo nuevas funciones y mejorando las existentes, lo que hace la librería más robusta y fiable.

En la farmacia una de las mayores modificaciones se deriva de la necesidad de controlar la caducidad de los productos. Si bien el framework original contemplaba que hubiera productos perecederos y no perecederos, era necesario añadir mecanismos para hacer chequeos acerca de la caducidad, para comprobar al hacer una venta si el producto ha caducado, etc. Los cambios realizados hacen que el framework no pierda genericidad (pues se hicieron con cuidado de seguir dando soporte a productos sin fecha de caducidad) pero gana flexibilidad.

Otros cambios no fueron reflejados en el framework por ser detalles muy particulares de la aplicación. Es el caso de las ventas rápidas, donde manejamos las ventas de una manera diferente a la común en el resto de los almacenes. De todos modos, en caso de que algún programador quisiera usar ésta característica podría consultarla en la aplicación desarrollada.

Una amplia colección de casos de uso, diagramas de secuencia y algunas de las pantallas de la aplicación pueden ser vistas en los capítulos pertinentes de la documentación.

Tecnologías

Aquí podemos ver un resumen con algunas de las tecnologías usadas a lo largo del proceso de desarrollo:

- **Enterprise Architect** es la herramienta CASE usada para el diseño del framework y las aplicaciones. El programa permite ingeniería directa e inversa para C# (lenguaje usado en la implementación), así como soporte pleno para UML.
- **.NET Framework** se usó en las fases de implementación y prueba, usando como acabo de mencionar C#. En la documentación se incluye una pequeña comparativa entre .NET y J2EE, resaltando los puntos fuertes y débiles de ambas plataformas. Tanto el código como la GUI se desarrollaron usando Microsoft Visual Studio .NET.
- **Nhibernate** fue la librería usada para la persistencia de objetos contra una base de datos relacional. La librería está en fase de pruebas y dio bastantes problemas, pero se espera que próximamente ofrezca mejores prestaciones. La base de datos usada durante las pruebas fue Microsoft SQL Server.
- **Nunit** se usó para probar las aplicaciones.
- Se usaron **Servicios Web** para hacer públicas las principales funciones de las aplicaciones.

Conclusiones y futuras ampliaciones

Al ser una aplicación fundamentalmente de desarrollo, no hay demasiadas conclusiones que se puedan sacar. Aquí se exponen algunas de ellas:

- Los patrones proporcionan un buen modo de describir los frameworks, ya que los usuarios no querrán saber exactamente como funciona, sino en cómo resolver un problema particular. Aunque los patrones y el diseño en 3 capas explicado anteriormente ofrecen teóricamente muchos beneficios, ha sido difícil comprobarlos, ya que los creadores del framework fuimos los mismos que lo usamos por primera vez, y ya conocíamos el modo de uso y funcionamiento de éste. Por tanto es necesario que futuros proyectos sean los que comprueben dichos beneficios.
- Usando MSTORE, el tiempo de desarrollo de las aplicaciones se vio decrementado en un tiempo aproximado del 50%, teniendo en cuenta que al escribir las aplicaciones también se realizaron tareas de depuración del framework. De todos modos para realizar dicha afirmación también sería necesario que éstas fueran desarrolladas por alguien ajeno al desarrollo de la librería.
- Los patrones son una herramienta potente en adición a un lenguaje orientado a objetos como Java o C#. Las diferencias entre ambas plataformas no son muy notables, en cualquier caso ningún lenguaje oculta la importancia de un buen diseño.

Por otra parte, los siguientes pasos en el desarrollo de MSTORE pasarían por:

- Reparar los posibles fallos en la implementación actual, ampliando los servicios web ofrecidos.
- Desarrollo de más aplicaciones que exploten las posibilidades de la librería y la amplíen.
- Integración con los programas paralelos desarrollados: gestión de riesgos y de documentos.
- Realizar pruebas con empresas reales y evaluar puntos fuertes y débiles.

Table of Contents

OBJETIVO ..... ***VI***

CHAPTER 1. INTRODUCTION ***1***

1.1 APPROACH OF THE PROBLEM 1
1.2 JUSTIFICATION 1
1.3 OBJECTIVE 2
1.4 PROJECT SCOPE 2

CHAPTER 2. ANALYSIS ***5***

CHAPTER 3. PREVIOUS STUDIES ***7***

3.1. REQUISITES OF THE SYSTEM 7
3.2. OBJECT ORIENTED FRAMEWORKS 9
3.3. JHOTDRAW 10
3.4. CONCLUSIONS 11

CHAPTER 4. FRAMEWORK DESIGN: PROBLEMS FOUND AND THEIR SOLUTIONS USING PATTERNS ***13***

4.1 ARTICLES MANAGEMENT 14
4.2 DOCUMENTS 17
4.3 STOCK MANAGEMENT 21
4.4 STORE MANAGEMENT 24
4.5 MOVEMENTS 31
4.6 FRAMEWORK DESIGN AND CONCLUSION 32

CHAPTER 5. APPLICATIONS DEVELOPMENT USING MSTORE FRAMEWORK ***33***

5.1 COMMON TOPICS IN A STANDARD STORE MANAGEMENT APPLICATION 33
5.1.1 Analysis 33
5.1.2 Design 44
5.1.3 Implementation 50
5.2 PHARMACY: WORKING WITH PERISHABLE PRODUCTS 51
5.2.1 Analysis 51
5.2.2 Design 58
5.2.3 Implementation 59

CHAPTER 6. USED TECHNOLOGIES ***63***

6.1 .NET FRAMEWORK 63
6.2 NHIBERNATE 68
6.3 EA 69
6.4 NUNIT 70
6.5 WEB SERVICES 71

CHAPTER 7. CONCLUSIONS ***73***

7.1 FUTURE PLANS 74

REFERENCES ***77***

APPENDICES ***79***

I. STORAGE SYSTEMS AND PRODUCT LOCATION IN A STORE 79
II. STOCK MANAGEMENT POLICIES 82

List of Figures

Figure 1: MStore project scope 3
Figure 2: Implementation level of an application using the Framework 5
Figure 3: The three design levels in the abstraction "Article" 6
Figure 4: Framework package diagram, first iteration 8
Figure 5: JavaDraw is a typical application of JHotDraw 10
Figure 6: Simple and Composed Articles example 14
Figure 7: Composite Pattern 15
Figure 8: Composite Pattern in simple/composed articles problem 15
Figure 9: Example about families and recursive families 16
Figure 10: Adopted solution for families' management 16
Figure 11: Builder Pattern Structure 17
Figure 12: Builder Pattern applied to the Document Types Management problem 18
Figure 13: Abstract Factory Pattern Structure 19
Figure 14: Abstract Factory solution 20
Figure 15: Observer structure 22
Figure 16: Strategy Pattern structure 23
Figure 17: Strategy solution 23
Figure 18: Zones in a store 24
Figure 19: Store management 25
Figure 20: Factory Method Pattern 27
Figure 21: Locators 29
Figure 22: Store physical distribution diagram 30
Figure 23: Factory Method pattern solution 30
Figure 24: Life cycle of the product in the store 31
Figure 25: Observer Pattern solution 32
Figure 26: Manage Company Use Case Diagram 34
Figure 27: Manage Resources Use Case Diagram 36
Figure 28: Manage Transactions Use Case Diagram 38
Figure 29: Manage Merchandise Use Case Diagram 40
Figure 30: Manage Store Use Case Diagram 41
Figure 31: Equipment Sequence Diagram 44
Figure 32: New buying sequence diagram 45
Figure 33: New Entry Sequence Diagram 46
Figure 34: New Output Sequence Diagram 47
Figure 35: Equipment solution – Factory Method 48
Figure 36: Transactions: Observer pattern solution 49
Figure 37: Resources management Screen- Choice: New Employee 50
Figure 38: Log in screen in the Ironmonger 50
Figure 39: Log in Screen in the Pharmacy 50
Figure 40: Common Use Cases that were modified in the Pharmacy 52
Figure 41: Modified Manage Merchandise Use Case for the Pharmacy 54
Figure 42: Specific Use Cases Diagram of the Pharmacy 55
Figure 43: Manage statistics Use Case Diagram 56
Figure 44: Manage Articles Use Case Diagram 57
Figure 45: Create Perishable Article Sequence Diagram 58
Figure 46: Fast Output Sequence Diagram 58
Figure 47: Main Screen of the Pharmacy Management application 59

Figure 48: Add article to entry screen 59
Figure 49: Set Stock Management Policy parameters screen 60
Figure 50: New Output screen 60
Figure 51: Pharmacy Web Services, mainframe 61
Figure 52: Invocation of one method and result 62
Figure 71: .NET Framework Architecture 63
Figure 72: .NET Framework Classes Library 64
Figure 73: Visual Studio .NET appearance 65
Figure 74: Logic behind a Web Service 71
Figure 75: Chaotic Storage 79
Figure 76: Dynamic by gravity-FIFO 80
Figure 77: Rack 80

List of Tables

Table 1: Actors and Goals 33
Table 2: Manage Company Use Case 34
Table 3: Manage Resources Use Case 35
Table 4: Manage Transactions Use Case 37
Table 5: Manage Merchandise Use Case 39
Table 6: Manage Store Use Case 41
Table 7: Manage Users Use Case 42
Table 8: Log in Use Case 42
Table 9: Log out Use Case 43
Table 10: Pharmacy: Manage Merchandise Use Case 54
Table 11: Pharmacy: Manage Store Use Case 55
Table 12: Pharmacy: Manage Company 55
Table 13: Pharmacy: Manage Statistics Use Case 56
Table 14: Pharmacy: Manage Articles Use Case 57

Acknowledgements

We would like to express out gratitude to the following people for the support and assistance in developing of this Master Thesis:

Our Supervisor in Oviedo (Spain), responsible for the idea and also Director of the entire Project that includes the rest of the modules that will work together for providing to the SMEs a good and complete tool that allow them their modernization and why not, creation of jobs in our small region. This can't remain as a dream!

Our supervisor in Luleå (Sweden), thank you for listening to our ideas, reading our long (and sometimes difficult to read) reports, providing a good place to work and showing us that Swedish kindness!

Our families, this time a little far from us, but always near in our feelings.

Our "family" here, in Luleå, which means our friends, who gave us the necessary support after the daily work.

And anybody we missed who deservers a mention!

Chapter 1. Introduction

1.1 Approach of the problem

This project is part of a software components creation program for SMEs (Small and Medium Enterprises), made in Oviedo University (Spain) [15]. This program is composed by several projects such us documents management, risk management, and the one is being explained in this document, store management. It is planned the development of future components, as an accountancy program, and expansions of the projects that are started now.

The aim of all these projects cooperating together is to provide the SMEs with attractive and viable software tools that allow their modernization for reaching the competitively grade of the department stores.

In the particular context it is talking about, MSTORE tries to be a framework capable to support the store management part. The library, composed by three layers, gathers the more general and common characteristics to this kind of enterprises characteristics.

Once "solved" the problem with MSTORE Framework, it should be probed its correct operation by means of the realization of two more evaluation projects, each one with its particular problems:

- ✓ Pharmacy Management
- ✓ Ironmonger Management

These projects are not supposed to be big and complex applications for the store management; they are only attempts to test the most important characteristics of the framework.

1.2 Justification

Asturias is one Spanish region with not much industry and not many job opportunities. If it is had in account also the national or international environment, each time more open, the enterprises are doomed to fortify and become more competitive.

As people living there can't stay sitting while they wait for a policy with multimillionaire inversions that help the sector, this project tries to contribute with some free tools, so the SMEs will be able to compete with the big companies that have more access to resources and knowledge.

The technology it is tried to contribute doesn't have to be understood as a support to the business traditional process, but the end that causes intellectual and human capital and enterprise formation and innovation in the information technology field.

1.3 Objective

In this project it is tried to define a software components library oriented to the region's small and medium enterprises (SMEs). As a whole, it is going to be defined the hierarchy, formal representation and implementation of each component. From here can be extracted the general and specific objectives for the project.

General Objective: To define a framework for helping the SMEs in the store management labour. The framework must be generic enough to be used in different types of stores, having into account the most important features of this kind of applications. Anyway it is important to stress that, given the wide variety between these types of programs; the framework will be valid for approximately 90 % of them, having into account that the remaining 10 % have specific features that cannot be seized into the framework. It is necessary a delimitation of the problem, so a generic framework can be built in a reasonable time. The problem is that removing flexibility also reduces reuse. It is expected that future versions will increase it, as it will be explained in next section.

For work management purposes, this general objective can be broken into four specific objectives:

- ✓ **Specific Objective 1:** To define the more relevant concepts for the formal description of the framework. This objective implies a search between the more common store management applications, trying to capture as many concepts as possible and writing down the common behaviour. The product of this objective should be a general requirement specification.
- ✓ **Specific Objective 2:** To make a generic model with the requirements specification. The design should be easy to understand for the developers who want to use the framework. Design patterns can be very helpful to make the design generic and usable.
- ✓ **Specific Objective 3**: To represent formally the framework implementing the more general part, common to all the stores. The objective includes the refinement and implementation of the design made. It will be fulfilled when the two first levels of the application are implemented and tested. It also includes a deep documentation of the classes and packages, including design patterns used.
- ✓ **Specific Objective 4**: To expand the framework, basing in concrete cases (ironmonger, pharmacy). These applications are not supposed to use the entire framework since it is only wanted to probe concrete parts of it. It is also wanted the framework to be extended with web services for the most important functions and some database facilities.

1.4 Project Scope

After these objectives, this project is closed but the idea is that another one or two projects will continue with it, in order to make it really usable. So, even if the design tries to capture as many features as possible, the particular applications will take a part of it and test it deeply. Next projects will try to amply the design (looking for missing features) and test the remaining part, as can be seen in Figure 1:

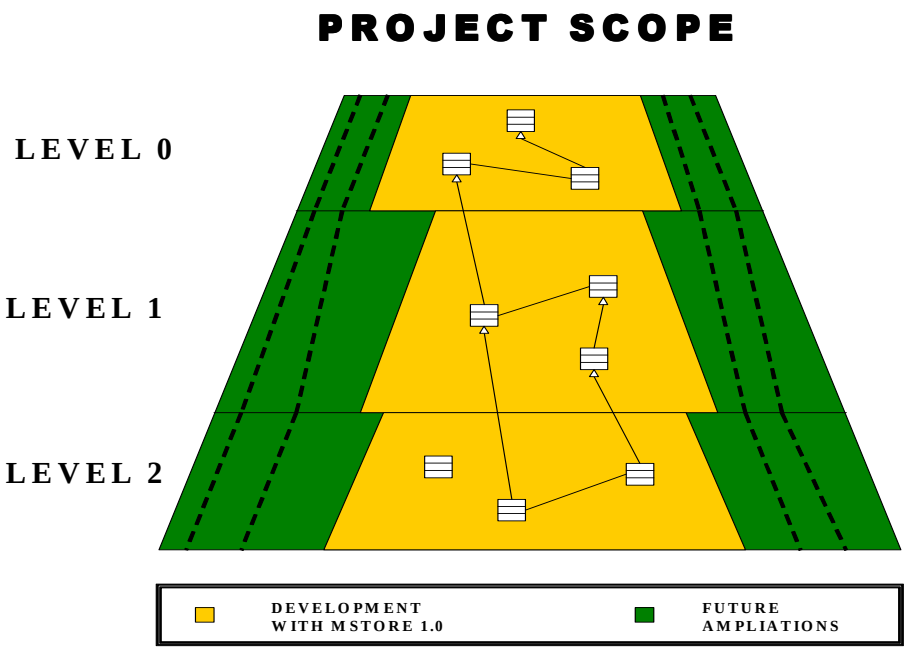


Figure 1: MStore project scope

Chapter 2. Analysis

As it was told before, in a very high level the system should have at least the next elements:

- **Level 0:** Interface-based framework that defines the basic behaviour of the system and its interactions without any concrete implementation.
- **Level 1:** Basic implementation of some behaviour that is considered common to any implementation. For instance the order management politics based in different quantitative methods.
- **Level 2:** Concrete Implementation of a client's solution with his concrete store operating policies etc.

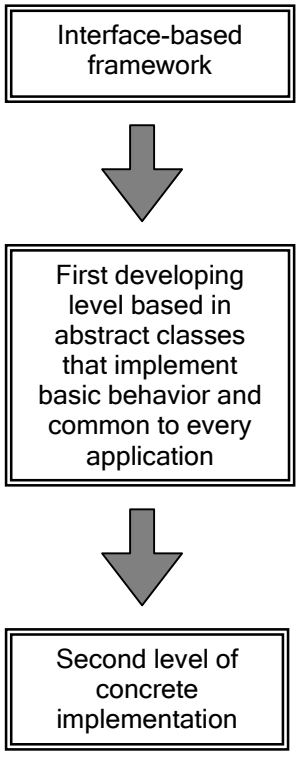

Interface-based framework
First developing level based in abstract classes that implement basic behavior and common to every application
Second level of concrete implementation

Figure 2: Implementation level of an application using the Framework

As the framework is trying to be as generic as possible, the abstraction of the entities must be independent of any implementation detail. The separation in two levels (interfaces and abstract classes) makes easy the differentiation of the framework design and its implementation. The 3rd level has also value as it is going to test that the framework works, and it provides some examples to the programmer who wants to use the framework for the first time.

This is not a new schema since JHotDraw [10] has used it for graphic applications design with probed benefits.

As an example it can be seen in the next Figure how to design each one of these elements for the real-world abstraction "Article" following the explained method. The interface *IArticle* is going to contain the abstraction. A big effort must be put in the design of the interfaces, since the relationships between interfaces in this level should never change. In the second level the abstract classes are implemented with the behaviour common for a set of articles. For example the abstract class *AArticle* can include all the characteristics common for all the articles, regardless of the kind of article. A concrete user in a particular application uses a concrete class from the third level, for example, *PerishableArticle*. It is possible also to create new classes adapted to a concrete case. It can reuse the behaviour provider by AArticle or redefine the operations for its purposes. It can also define new operations or implement some provided by the interface.

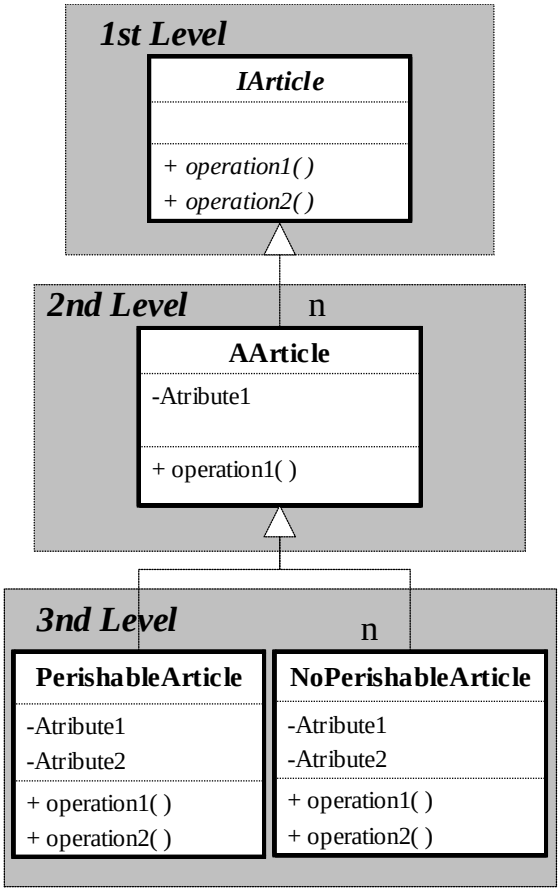


Figure 3: The three design levels in the abstraction "Article"

It is going to be used this name convention along the entire framework:

- **Interfaces:** < I > plus the name of the interface.
- **Abstract classes:** < A > plus the name of the abstract class.
- **Concrete classes:** Simple name of the class.

It will be also tried to give simple names for the abstractions in 1st level, and more complex names if necessary for the subclasses found in lower levels.

Chapter 3. Previous Studies

3.1. Requisites of the System

First studies on the store management field show that it is a very heterogeneous problem.

Companies interested on a program can have different needs depending on many factors: the type and size of the company, the articles managed, the processes occurring in the store, the concrete layout distribution, etc. [2][7]

A generic framework that covers all kind of stores would take a huge amount of time developing it. It would also become very complicated, as the mechanisms to make it generic would turn much bigger. However, if the problem is delimited and a small number of special applications (no more than 10%) are let out of it, a general framework can be made that covering 90% of the usual store problems and turns out to be much more easy to develop and use.

Thus a general overview into the system would show that the store management framework should take into account the next points. This is only a general approach, more detailed requisites list have been developed but they are not included in the documentation:

- **Articles:** Stock management control, codes management, modelling of perishable articles (handled by sets and by sell-by date) and non-perishable articles (handled individually), articles managed by sets, ranges, families, catalogues, prices and benefit control, handled units control.
- **Merchandise entries and leaving:**
    - Management of everything concerning to product **entries** in the store: delivery zones, products contrasted by delivery notes, to layout in the store shelves with indication of the routes to use in the articles layout.
    - Management of **product leaves** in the store: order preparation management, packer management, products pick up management (minimizing the movements to the shelves), etc.
- **Order management:** Stocks management based in traditional methods (minimal stock) and the classic stock management policies. Buying to the providers, selling to the clients, and management of the different possible documents (budgets, delivery notes, tickets, invoices…).
- **Movements:** It is allowed stock changes between zones and also between stores... and the products kept in the stores. It is important to keep all the information about these movements because it is mandatory to know in all the moments the state and location of all the products in the company (inventories…)
- **Resources management:** Employees, machinery, costs and time control…
- **Store:** Controls everything related to stores (implementation must be multi-store)

  - **Layout:** It provides support for different types of equipment like Corridor Distribution.
  - **Distribution:** It is necessary to have some methods for locating free places in the entries, and do the symmetric process in the exits. These methods can be different depending on the zone. For example can be useful to use a FIFO method when the zone works with perishable products.
  - **Physical Distribution**: methods for making graphical representation of the store distribution. This will help the user of the future application to have control of the entire layout easily.
- **Web services:** All the main operations should have an interface as a Web Service, so that there should be done an MVC architecture that allows accessing the main application functions with different interfaces, between them the Web Services.
- **Database management:** Object-relational layer (database access broker) that encapsulates the access.

In a first iteration, the simplified package diagram of the framework could be as follows:

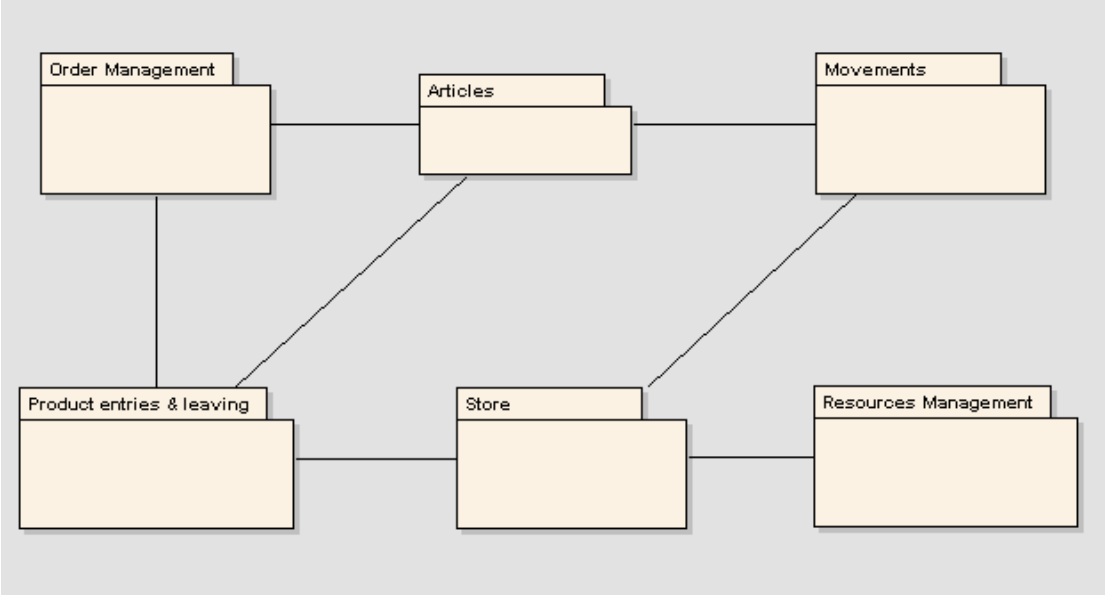


Figure 4: Framework package diagram, first iteration

The design of the framework should take into account all of these points, making the design generic enough to extend them when necessary. The reader could think that the purpose of the project is just the implementation of a standard application management application, but instead of that what is going to be built is a generic design so that the developer can change the parts of it that doesn't like (or doesn't fit to his necessities), and adapt in the way he wants, which is much more flexible than a single program. The characteristics and benefits of the frameworks are discussed in next section.

3.2. Object Oriented Frameworks

There are many ways to define a framework. One generic definition could be the following:

> *"A framework is a reusable design of a program or a part of a program expressed as a set of classes" [6]*

For the problem it is tried to solve, this definition looks like more suitable:

> *"A framework is a collaboration of adaptable classes that define a solution for a given problem"*

Here follows its desired characteristics:

- It defines the main abstractions and their interfaces.
- It establishes the relationships between the objects.
- The framework is adapted to particular solutions using the redefinition.
- It should add any default solution.

Patterns are very related with the Object Oriented Frameworks since both Patterns and Frameworks facilitate reuse by capturing successful software development strategies.

The main difference is that frameworks focus on reusing concrete designs, algorithms, and implementations in a particular programming language. In contrast, patterns focus on reuse of abstract designs. The pattern says HOW solve a problem, while the framework gives a SOLUTION (or even *is* a solution).

Some of the main benefits of Object Oriented Frameworks are explained next: [25]

- ***Modularity***: Frameworks make an explicit differentiation between design and implementation using interfaces and abstract classes. The interfaces are stable, while the abstract classes encapsulate volatile implementation changes. It is also easier to calculate the cost of changing some parts of the design and the implementation, reducing the effort required to understand and maintain the software.
- ***Reusability***: The interfaces define generic components that can be used to create new applications. As they are already created and validated, developers can use the previous efforts in order to create their own designs. The reusing improves the programmer productivity, as well as the quality, performance and interoperability of the software.
- ***Extensibility***: A framework provides explicit hook methods that allow applications to extend its stable interfaces. Hook methods decouple in a systematic way the stable interfaces and behaviours of an application from the variations required by instantiations of an application in a particular context.

Since a Framework is software, it is a mixture of the concrete and the abstract. And since frameworks are reusable designs, not just code, they are more abstract than most software, which makes documenting them difficult. Frameworks are designed by experts in a particular domain and then used by non-experts. The principal audience of framework documentation is someone who wants to use the framework to solve typical problems, not someone building a software cathedral. Patterns are well suited for this audience.

It is widely accepted the convenience of **documenting** frameworks with patterns [3] [14]. The main purpose of a set of patterns is to show how to use a framework, not to show how it works, but patterns can also describe a big part of the theory of its design. Each pattern describes a problem that is common in the problem domain of the framework, and then describes how to solve that problem.

Each pattern has the same format. The more used format is to first give a description of the problem. This is followed by a detailed discussion of the different ways to solve the problem, with examples from other parts of the framework. The pattern ends with a summary of the solution. Patterns are problem oriented, not solution oriented. Each pattern describes how to solve a small part of the larger design problem. Sometimes the solution is to make a new subclass, sometimes it is to parameterise an object of an existing class, and sometimes it requires connecting several objects together.

Documentation for a framework has three purposes, and patterns can help fulfil each of them. It must describe:

- ✓ The purpose of the framework
- ✓ How to use the framework
- ✓ The detailed design of the framework.

Patterns are best suited for teaching how to use a framework, so a set of patterns can meet all three of these purposes for framework documentation.

3.3. JHotDraw

JHotDraw [10] is a good example about of using patterns for the description of a framework. It is a highly customisable GUI framework that simplifies developing drawing applications. It is inspired by *HotDraw*, developed by Kent Beck and Andre Winand, and it was developed by Thomas Eggenschwiler and Erich Gamma and presented in OOPSLA97. It was created for a seminary as an example of patterns application in frameworks creation, but its ideas can be directly applied to professional applications.

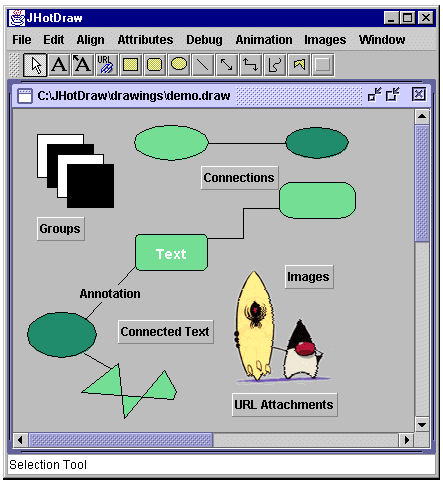


Figure 5: JavaDraw is a typical application of JHotDraw

JHotDraw defines a basic skeleton for a GUI-based editor with tools in a tool palette, different views; user defined graphical figures and support for saving, loading and printing drawings. The framework can be customized using inheritance and combining components. Design Patterns and a programming platform like Java (and, as the reader will see later, .NET also) are a very powerful combination, although no language can reduce the design importance.

The framework is very interesting from a software engineering point of view. With some knowledge of JHotDraw structure, it can be extended to include missing functionality or to change existing one. It is one of the first software development projects explicitly designed for reuse and labelled as a *framework*. It was also documented very early in terms of design patterns, and therefore was very influential to the design pattern community. It is composed of fundamental design patterns, such us Composite, State, Template Method, Factory Method and Strategy. Knowing the basic concepts behind those patterns, it is an easy task to adapt the framework to meet the particular application's requirements.

3.4. Conclusions

From the objective of developing a tool that could make easier the task of creating store management applications, it seems logic to start looking for information about **Store Management specifications** with the aim of understanding the problematic of managing a store. After some studies that showed how wide is this field, all the specifications were delimited and divided between two students, each in charge of a set of packages. Both students worked in parallel and collaborated to build the whole framework together.

At the same time, doing this exercise, the designers would know the problem in **depth**, and they would be able to face up the carrying out of the framework MStore better now. This framework will provide to the user with the benefits pointed in the third section of this chapter, and give to the reader a lot of advantages in the store management field.

Before doing this, to make a management program could seem an easy task, but if the size becomes bigger, it is necessary to think in other solutions that help the designer (design patterns, using of Frameworks, technologies…).

So, after that, it was very useful studying **O. O. Frameworks** more in depth, before starting one. The chosen one was **JHotDraw.** In it, the patterns established the design structure.

Applying **design patterns** there will be achieved a **solid solution** for each problem raised during the analysis of the program, with the benefits that the design patterns provide to us, since they have been probed a lot of times.

Learning one Framework, and also MSTORE requires **a bigger initial effort**, until understand it completely, but then it **reduces development time** and improves the quality of the software.

This learning time is largely decreased if the Framework has a **good documentation**. This means that it has comments and descriptions about its design, and also good user manual. Using standards helps to the user also.

The researchers tried to apply all of these characteristics as basic rules for the development of their system. Anyway it is not always automatic to know what pattern to use. The implementation of the pattern is an easy task, but the knowledge about when and where using them is more difficult. Sometimes some of them are applicable at the same time. As will be seen in next chapter, it also happened when developing **MSTORE**.

Chapter 4. Framework Design: Problems found and their solutions using Patterns

The general problem is to face up the management of one multi-store, which is more complex than the reader can think in a first view. But it is also very repetitive in all the stores since they usually have many things in common.

It is wanted to make a general Framework called MSTORE that collects all the support, as far as possible, for making a typical store management application.

Before to do this it must be raised what problems can be found, which are the possible solutions and, of course, the best solution for that problem. So there will be followed this outline in this report.

Design Patterns [9] represent recurring solutions to software development problems within a particular context, and they can help us to resolve our problems. Before starting to read it is useful to know more about Design Patterns and Stores, in the References there can be found several links and books.

Frameworks can be understood as a concrete reification of families of design patterns that are targeted for a particular application-domain. Likewise, design patterns can be viewed as more abstract micro-architectural elements of frameworks that document and motivate the semantics of frameworks in an effective way. When patterns are used to structure and document frameworks, nearly every class in the framework plays a well-defined role and collaborates effectively with other classes in the framework.

Following chapters explain some of the problems found in the different sections of the framework and the solution given, expressed as a design pattern if that is the choice, but not always the patterns are the best solution, and in that case, there is explained the reason. Before choosing a solution, there are postponed other possible solutions and explained why the selected one is the most suitable. When the solution is obvious, there are just explained the benefits of it. Sometimes the diagram doesn't fit 100% with the adopted solution (in order to make it more simple), but the examples are trying to reflect it as truly as possible. Some diagrams that need more explanations are also shown in the Framework Development Section.

Although the Framework include more packages, in this documentation there will be only commented the most important and in our opinion, more interesting problems in an academic approach.

4.1 Articles management

PROBLEM 4.1.1: COMPOSED ARTICLE TYPES

The framework that is being designed must be adaptable for all kinds of stores that can deal with a long variety of article types. For instance, simple stores will only deal with raw articles, and the merchandise they receive is the same that they are going to sell. A good example could be a simple bookshop store, where the books are received, stored and then sent away in the same shape.

More complex stores are susceptible to deal both with simplex and composed articles. For instance, a computer store could want to deal only with single components (keyboards, displays, memories, etc.) or with more complex components that were created in the store (for instance the whole computer). These complex components can share some characteristics with the simple articles or can differ in others (like the price in the example). Anyway it is a good that the complex article knows all the information about the simple articles. The figure 6 illustrates the example.

It is also very desirable that can be worked with both the complex and the simple articles in the same way, but adding some mechanism that allows knowing whether an article is simple or composed.

One keyboard is an article. It is composed by only one simple article.

One personal computer is an article also, but it is composed by other components such as mouse, keyboard, etc.

Figure 6: Simple and Composed Articles example

Summarizing, it is wanted the next behaviour:

- ✓ The articles can be simple or many simple articles can compose them.
- ✓ It is wanted to considerer simple and composed articles in the same way.
- ✓ It is wanted to access to the properties of the simple articles from the complex one.
- ✓ It is wanted to have recursive composition: one complex article composed by more complex articles.

SOLUTION:

- ➢ Composite Pattern

The **Composite Pattern** offers a quite-straightforward solution to the explained problem. The pattern composes objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and composition of objects uniformly and makes easy the access from

the composed class to the simple classes, so it fulfils all the desired requirements of the solution.

The key of the pattern is an abstract class that represents both primitive types and their containers (*Component*, as can be seen in Figure 7*)*. The class declares operations that all Composite objects share, such as operations for accessing and managing its children, and operations that all Leaf objects share (and Composite objects also if necessary).

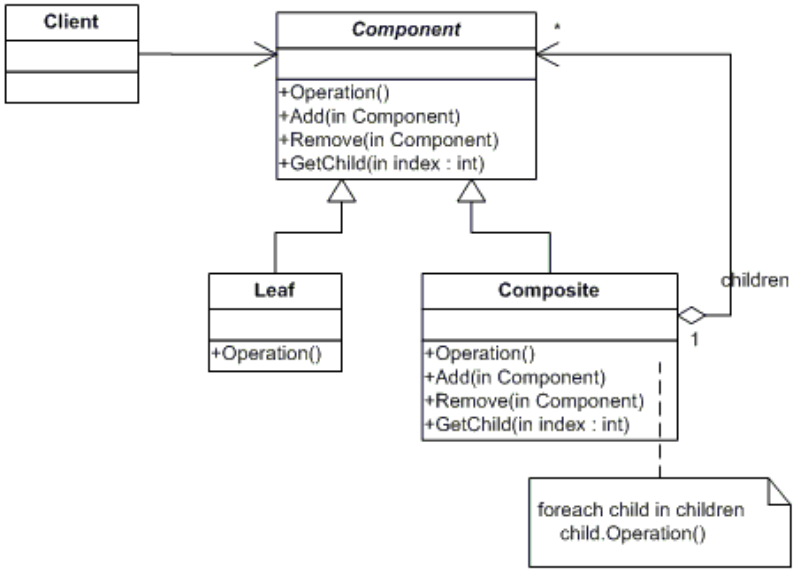


Figure 7: Composite Pattern

The application to our framework is made in the second level, as there are going to be implemented some of the operations from the Component class. The Figure 8 shows the translation into the framework classes. Not all the operations are reflected in the diagram, only a couple of operations are shown in order to illustrate the problem.

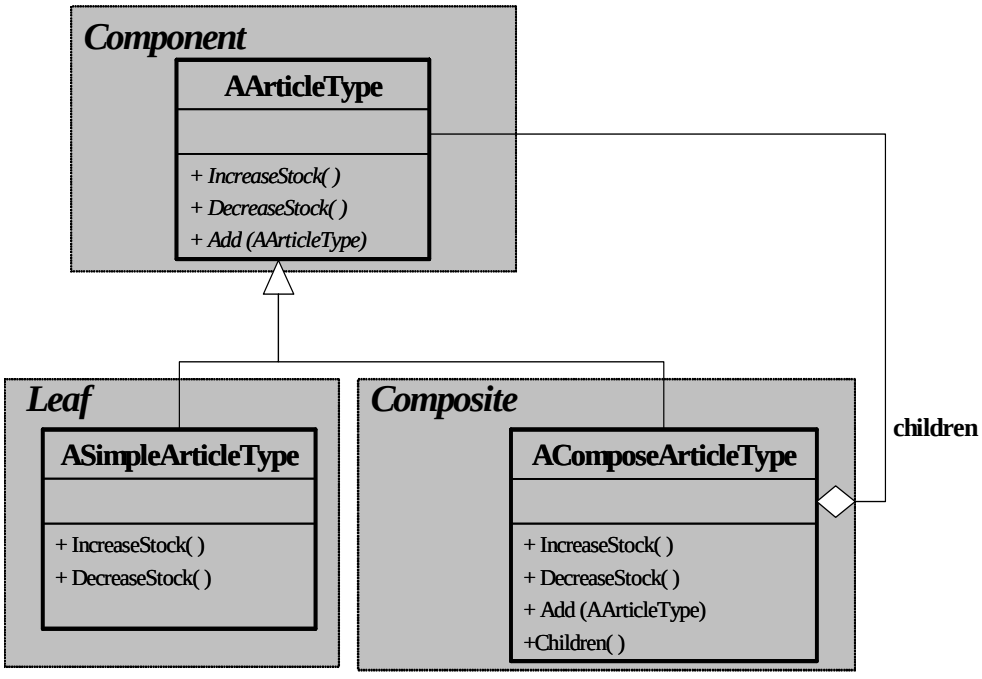


Figure 8: Composite Pattern in simple/composed articles problem

The client is neither reflected in the diagram, but it is represented by all the abstractions that use the article types: buying and selling are the ones that use them with more frequency.

This example reflects how one only pattern can solve several problems. In this case, the pattern fits perfectly in the suggested scenario.

PROBLEM 4.1.2: FAMILIES

Also working with the article types, it can be found that many companies can desire to group their article types into families, so that it is easier to manage them. These families can be composed of some articles or can be composed of more families. The next example helps to understand the concept:

As seen in the Figure 9, a family can be composed for more families or for fruits.

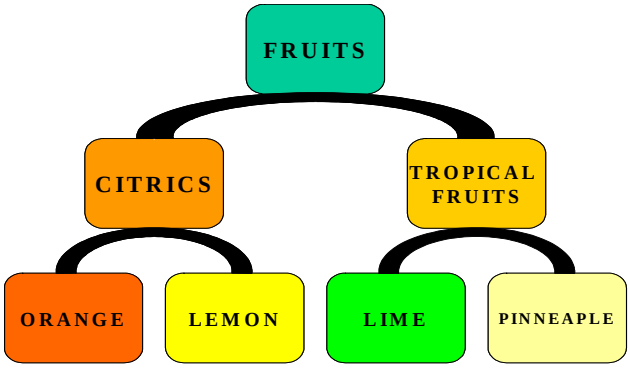


Figure 9: Example about families and recursive families

The problem is quite similar to the one explained about simple and compose article types, since it is wanted to get the recursive composition, but there is a key factor that makes it different: when dealing with the articles, the top-hierarchy class *AArticleType* was susceptible to be both a compose article type or a simple article type. Applying the **Composite Pattern** it can be used that class regardless it is wanted to use simple or complex articles.

If it was created a similar class (let's name it *AFamily*), the class should be susceptible to work both as a family and as an article type. But this is a wrong approach, since an article will never be a family itself (even while it is possible that families with *one* article exists, but this is different). So in this example the Composite pattern doesn't fit.

The solution this time will not use any design pattern, as the problem can be solved with simple composition and the use of any other pattern would be tiresome. The used solution is shown in Figure 10:

SOLUTION:

- Composition (no Design Pattern)

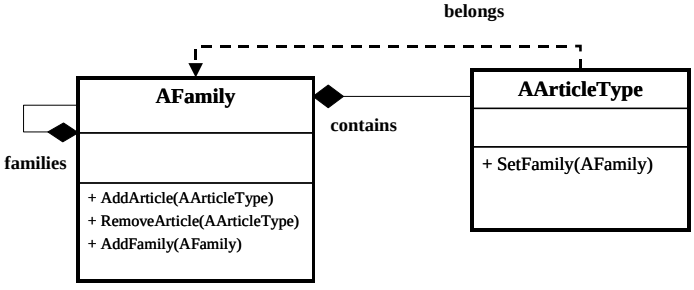


Figure 10: Adopted solution for families' management

This example illustrates that the design patterns are not always the best solutions, and, usually by complexity reasons, it is preferable to choose a simple solution than the application of a pattern.

4.2 Documents

PROBLEM 4.2.1: DOCUMENT TYPES MANAGEMENT

The document management is a delicate topic when talking about the store management. Every store uses to manage the documents in a different way, including different types of documents (depending on the company activities), different information for each kind of document and, of course different format.

Sometimes it is also desirable to be able to make a conversion between the different documents, as each one represents a different lifetime of a buying. For instance, in some stores, when the client wants to make the acquisition of some products, the first step is to ask the store for a budget, where there is an approximate price for the buying. If the budget is fair for the client, then he makes an order, with almost the same information than the budget but changing some details (i.e. the total price or some articles) and adding some other ones (like the paying method). Finally, when the order is delivered, it is added a document called delivery note that could have further information such us the day and the time of the delivery.

Summarizing, it is wanted a solution that evolves the following points

- ✓ The framework can deal with different kinds of documents: Budgets, Orders, Invoices, etc., all of them with different structure and information
- ✓ It is needed some efficient way to represent them.
- ✓ It is also useful some procedure to promote between documents (i.e. change from a Budget to a Invoice)

After analysing the problem, there were found two different solutions. As in a first iteration solutions looked to be equal (both with advantages and disadvantages), so it was made a full design for both solutions and then it was easier to decide which one to choose.

SOLUTION 1: Builder

The first solution is based on the **Builder Pattern**. The pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. The basic structure of the pattern can be seen in the Figure 11:

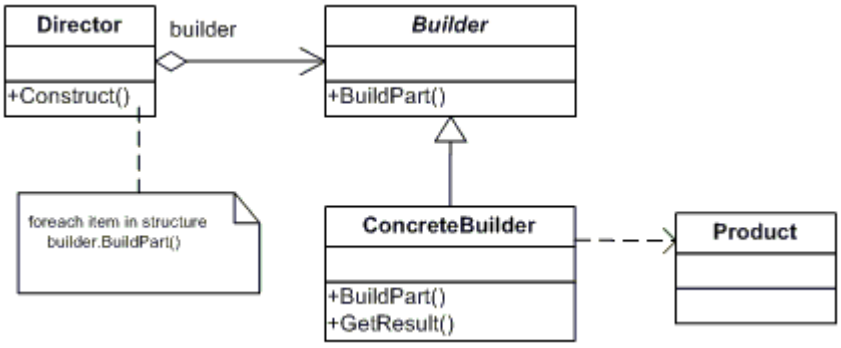


Figure 11: Builder Pattern Structure

The mission of the Director class is to create the Product. For this purpose, it is going to use a *Builder* class that is the class in charge of really creating the Product. The Builder interface allows the creation of the different parts of the product. If different kinds of products are wanted, different kinds of builders have to be created.

The pattern applied to the problem gives the next solution:

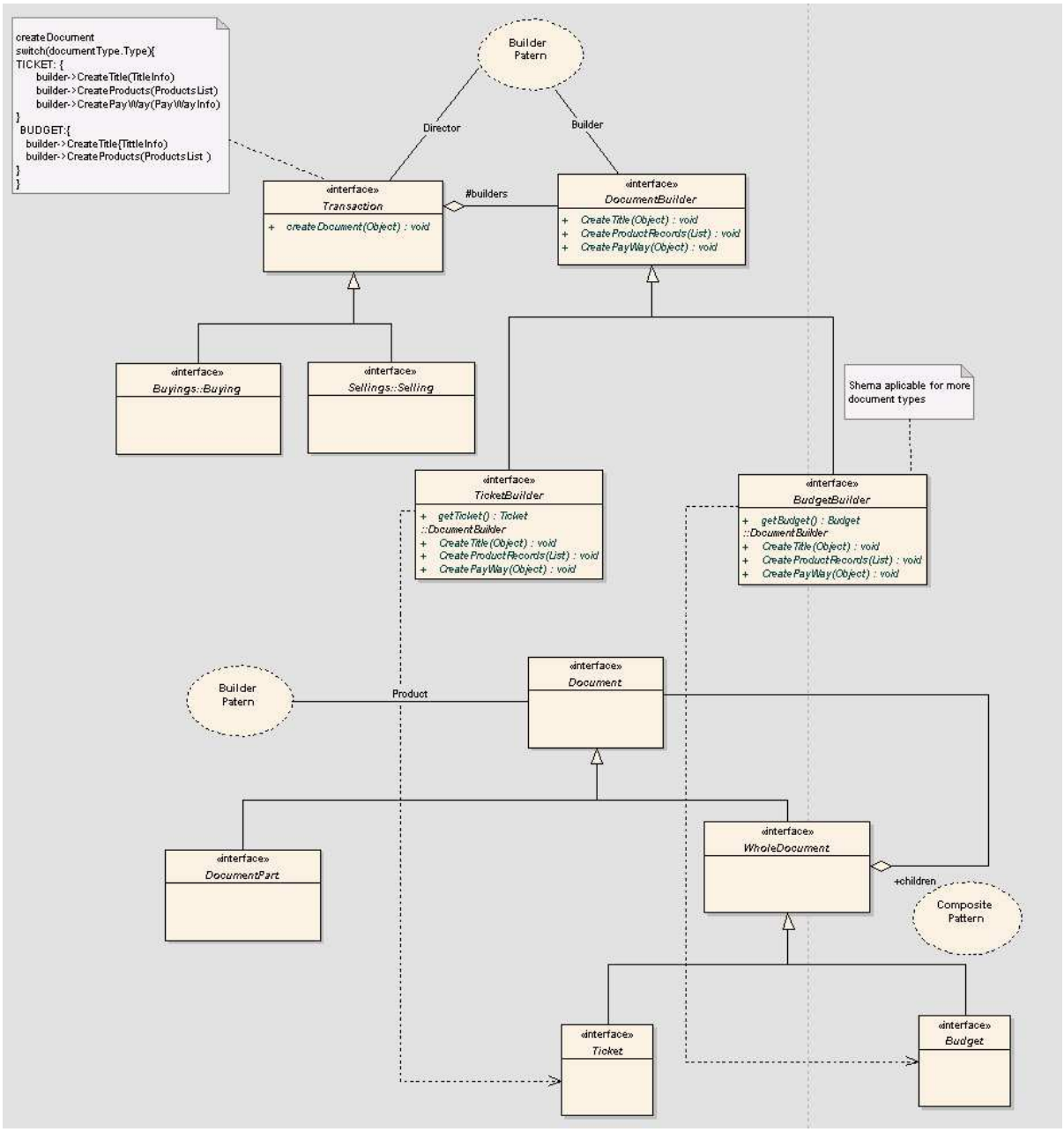


Figure 12: Builder Pattern applied to the Document Types Management problem

The Director is the class in charge of creating the document, usually a buying or a selling but it is possible that also another abstraction wants to create a document (it should always derive from the *ITransaction* interface). The director is going to possess a relation with the builder that is a class inheriting from the *IDocumentBuilder* interface. Depending on the desired document, the Director will instantiate a different concrete builder. After the creation of the builder, successive calls are made to it in order to create the parts of the document. In this

preliminary iteration, there is a Composite abstract document and the concrete documents inherit from it. Later it will be probed that this is not the best approach.

This solution makes easy the variation of the document internal representation, which is very good for the documents promotion as it is easy to share a common part for some of the documents. It also makes the representation of the document independent from the construction, so the transactions don't have to worry about how to build the document. It is also easy to create new kinds of documents.

SOLUTION 2: Abstract Factory

The second solution is based in the **Abstract Factory Pattern**. The pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. The pattern achieves that the transactions are independent from how are the documents created, composed and represented.

The structure of the pattern can be found in Figure 13.

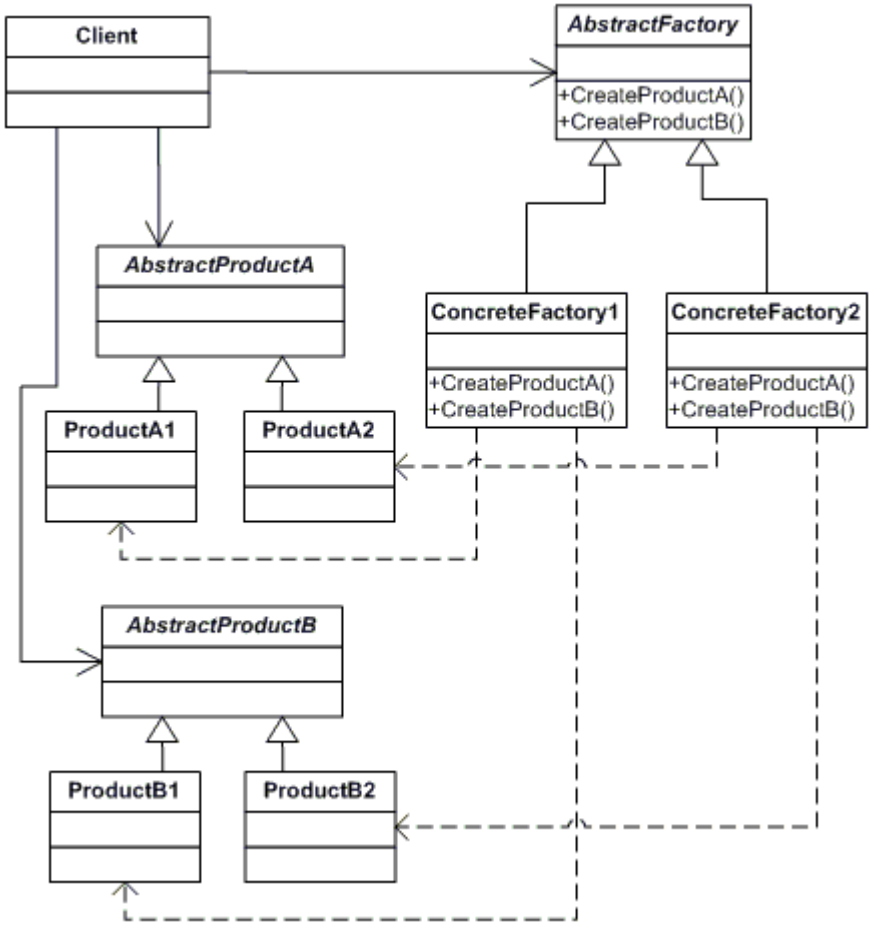


Figure 13: Abstract Factory Pattern Structure

The *AbstractFactory* declares an interface for operations that create abstract product objects. The *ConcreteFactory* classes implement the operations defined by the interface in order to create the products. The client will use the abstract products, but the factories are going to create concrete products that are subclasses of the abstract ones.

Applied to the concrete problem, the solution structure would look like the Figure 14:

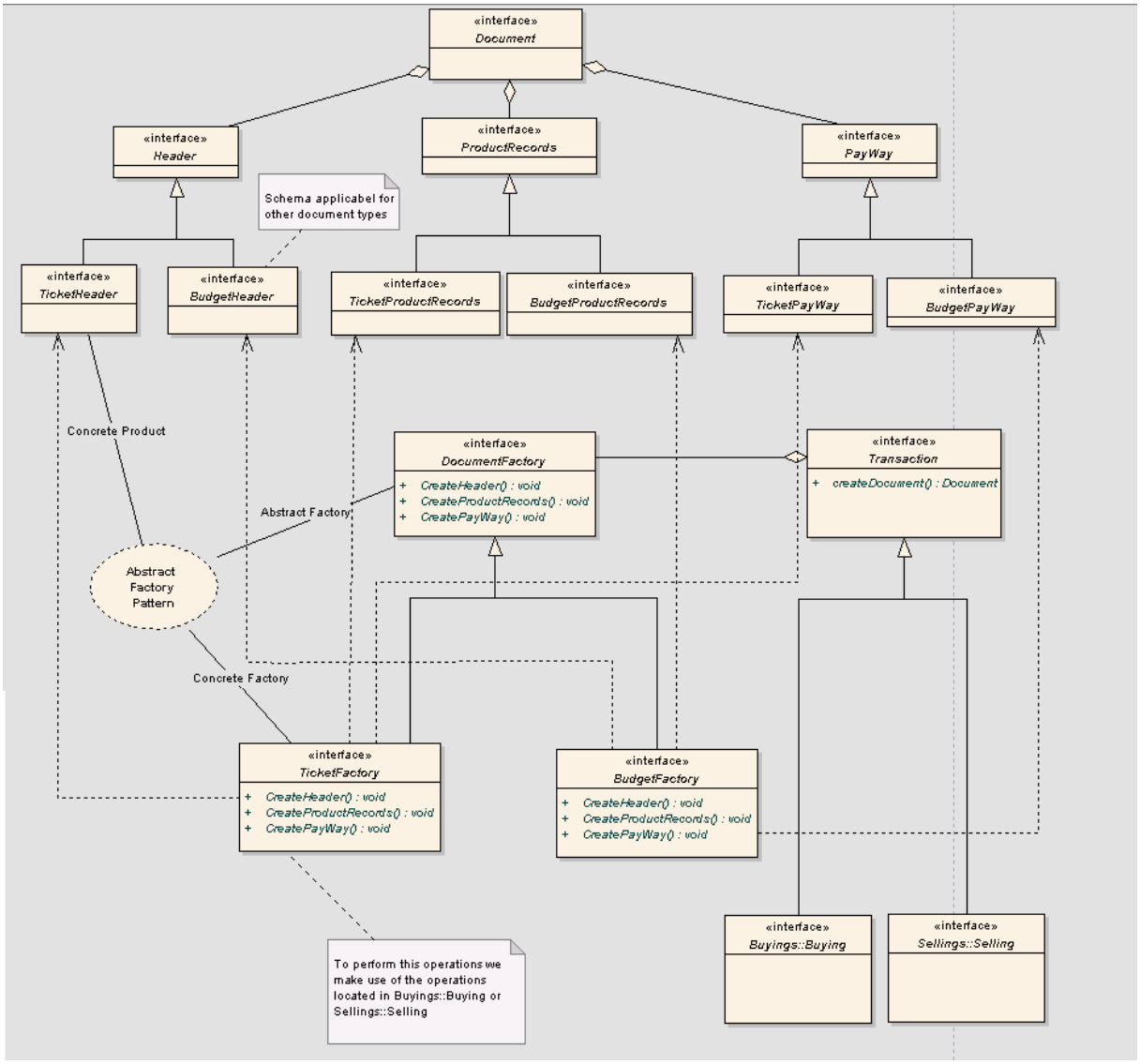


Figure 14: Abstract Factory solution

The *DocumentFactory* is the abstract factory, and it is going to be created a concrete factory for each different document that can be managed. On the other hand, some different parts will compose one document: a header, some records, the payment part, etc… Each of them will be represented by an abstract product, and will create a concrete product for each of the documents. The transaction will be the client, in charge of creating different documents.

The pattern makes easy to modify the structure of the document, as it is divided in some parts. It also makes easy to share the common parts between similar documents, and then it is only needed to create the different parts. Thus, the promotion between documents is also easy. However, the pattern introduces an important bad point: it is difficult to support new kind of products, so, even if it can give flexibility, the structure of the different documents is difficult to expand.

DECISION

This last point is definitive in the choice of the pattern to use. The Abstract Factory looks like a more flexible and powerful option, but it makes hard the addition of new parts in the documents, feature that can not be accepted for the use of the pattern in the framework, so it was decided to use the **Builder** pattern, which is also a good solution, even while it doesn't

make so easy the interchange between documents). Another point that helps the choice of this pattern is that the abstract factory looks like more complex, so it can be difficult to use with many document types and many parts.

A mixed solution, using both patterns, could be also a good way to solve the problem. However, the achievement of a good solution could be quite time-consuming, and it is out of the scope of this project. Anyway for further research it could worth trying to use it.

As it was suggested before, it is going to be made a change in the proposed solution, where the document was a composite. Further analysis in this point shows that it is not possible to apply this pattern, as a document part cannot be a document itself. So there will be used simple composition and inheritance when developing the document structure.

This example it is documented as a good example about how to decide which pattern use when there are different choices, evaluating the strong and the weak points of each pattern and weighting up them depending on the important requisites of the system. Sometimes it is necessary to sacrifice some good characteristics in benefit of another that are more important. Normally having a look on the pattern catalogue is enough, but many times a first iteration in the solution development is made in order to find the pros and cons.

The example also shows that the solution offered by the catalogue doesn't always fit to the problem, and it is needed to make small changes on it in order to make it suitable.

4.3 Stock management

PROBLEM 4.3.1: STOCK MANAGEMENT MANTEINANCE

Another key point in store management is the products stock management. One abstraction is going to be in charge of the stock count, but some other ones will make changes in this stock (for instance, the entries and outputs or products, or the buys and the sales can make changes in the virtual stock). It could be also wanted to maintain in the products another kind of stocks (for statistical or historic reasons). It is good to give the framework any mechanism to maintain this data with minimal work by part of the product, which shouldn't worry about these operations.

SOLUTION:

In the scope of this problem the **Observer Pattern** is very suitable to use. The pattern defines one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The key objects in this pattern are *Subject* and *Observer*. The subject is susceptible to have any number of dependent observers. All observers are notified when the subject changes its state. The structure can be seen in the Figure 15:

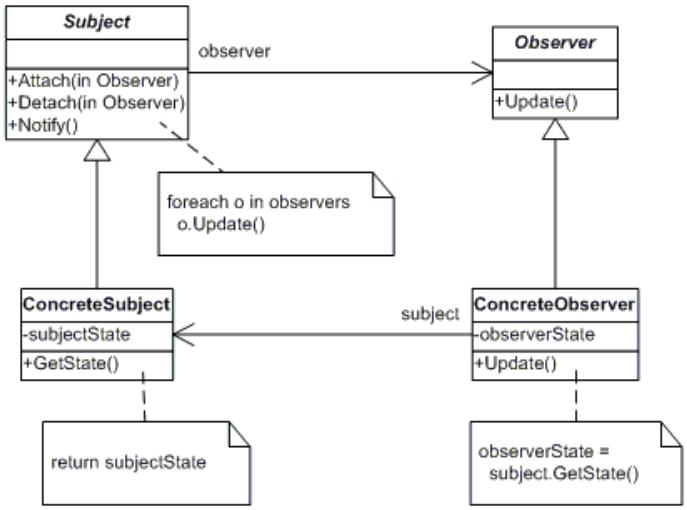


Figure 15: Observer structure

The *Subject* interface provides an interface for attaching and detaching *Observers*, and the *Observer* defines an updating interface for objects that should be notified of changes in a subject. The concrete observer maintains a reference to the subject, and it implements the updating interface to keep its state consistent with the subject one. The main concrete subject function is to store the state of interest for the concrete observer objects, as well as send a notification when its state changes.

It can look like a complicated solution for only one or two transactions, but makes easier the process when there are have many data to update or many subjects.

This solution shows how Design Patterns can help to make the system extensible. Sometimes problems can be solved with a solution "add hoc", but when designing a framework it is convenient to think that the system is going to be extendable. Patterns help in the task of making the programs easy to enlarge.

PROBLEM 4.3.2: STOCK MANAGEMENT POLICIES

One of the objectives of the framework is to include there the classic stock management policies (see Appendices). These policies are more used in theory than in practice (at least for SMEs), so it is a good practice to include them in the software and check if they are really used. All the policies are used for check if the stock level existing in the store in any given moment is enough or the store manager has to make a new order.

Even while all the policies have the same aim, they also have different parameters and characteristics (one algorithm that sometimes is shared by more that one policy). They are also invoked at different times. For example, one policy check that the stock is not under minimal after one selling has been performed. However, another policy makes this checking each T period of time.

Although these policies are different, they should be interchangeable when necessary. It is needed one design that allows easy variation of the policy, as well as independence between the policy and the product (the product should not care about the policy used).

SOLUTION:

The **Strategy Pattern** solves the problem it is raised. It defines a family of algorithms, encapsulates each one and makes them interchangeable. It lets the algorithm vary

independently from clients that use it. The pattern structure, shown in Figure 16, is simple but it makes very easy the reuse of the different strategies.

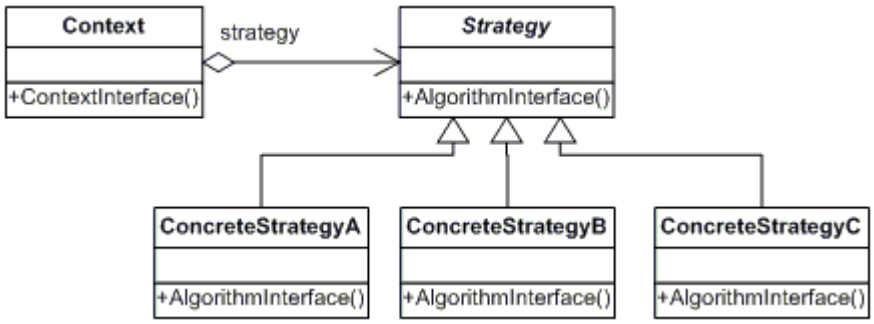


Figure 16: Strategy Pattern structure

The Context maintains a reference to a *Strategy* object. The *Strategy* declares an interface common to all supported algorithms. The Context uses this interface to call the algorithm. The *ConcreteStrategy* classes implement the algorithm using the Strategy interface.

Applied to our concrete problem, the pattern structure looks like in the Figure 17. The application of the pattern is quite straightforward:

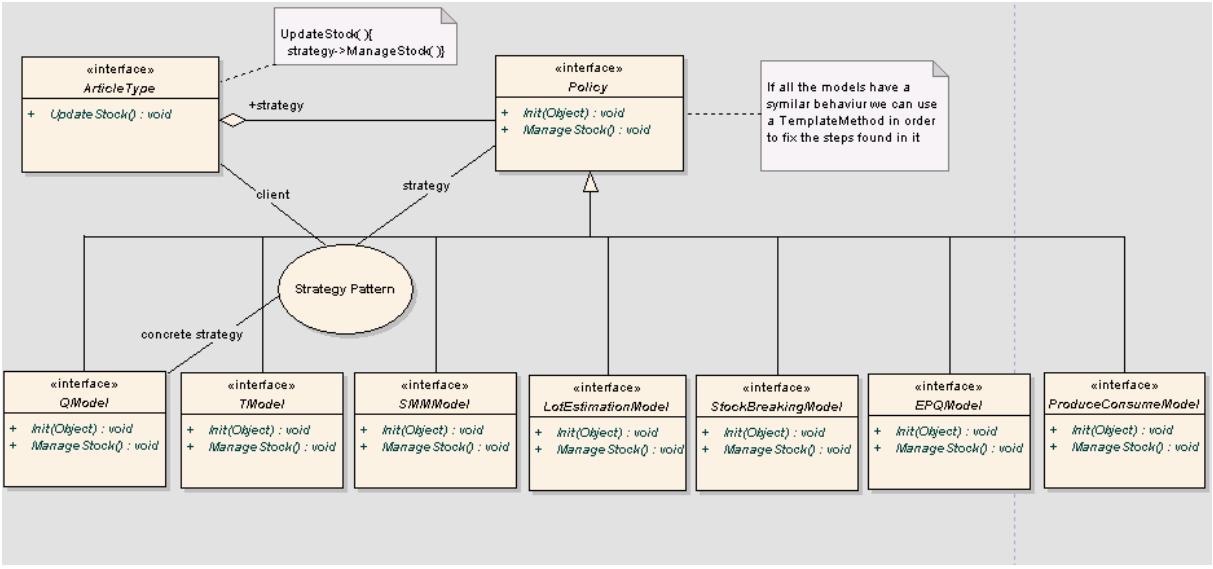


Figure 17: Strategy solution

The *ArticleType* class takes the Context role, and the *Policy* interface is the *Strategy*. The only operation that is going to be performed is the method *ManageStock()*, called when it is necessary to check if more stock is needed. The concrete strategies are the concrete policies that can be. With this pattern is very easy to add new policies, as well as interchange them when necessary. The article type doesn't care about the data used by the algorithm, just to init it.

It is not going to be developed the solution, but in early stages of the development it was thought the possibility of using the **Template Method Pattern**. It is not going to be explained in detail, but the pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. In order to decide between patterns, the pattern was applied to the problem (as happened with problem 4.2.1) and the pros and cons were weighted. This time

they were quite similar (this pattern allows implementing the common parts of the algorithm and leave it up to subclasses the implementation of the behaviour that can vary).

The decision this time was made in terms of complexity. After deeper study in the algorithms, it was discovered that they were not so complex as expected, so it didn't worth to use the Template Method pattern, thought for more complex algorithms.

This example shows how sometimes it is better to choose the simplest solution, even while in first iterations it can look like that the complex is the better.

4.4 Store Management

In this subsection, there are going to be discussed all the things related with the store, its distribution (layout of the physical store, locate places for new products or locate products in the store), and its movements between different zones or even other stores. As it is difficult to understand all like a whole issue, there must be decomposed in smaller problems:

PROBLEM 4.4.1: LAYOUT

- ✓ It is important to have the company organized. It can be interesting to know information only about some areas, or maybe to fix some parameters (costs…) in some particular areas. So there must offer the possibility of having nested areas.
  - ➢ The company should be distributed depending of the geographic situation, being possible to have hierarchies of areas. And each area will have one or more stores.
  - ➢ And also, each store will have different areas such unloading area or store zone. All of these areas usually have common behaviours, so they should inherit from one superior Zone class.

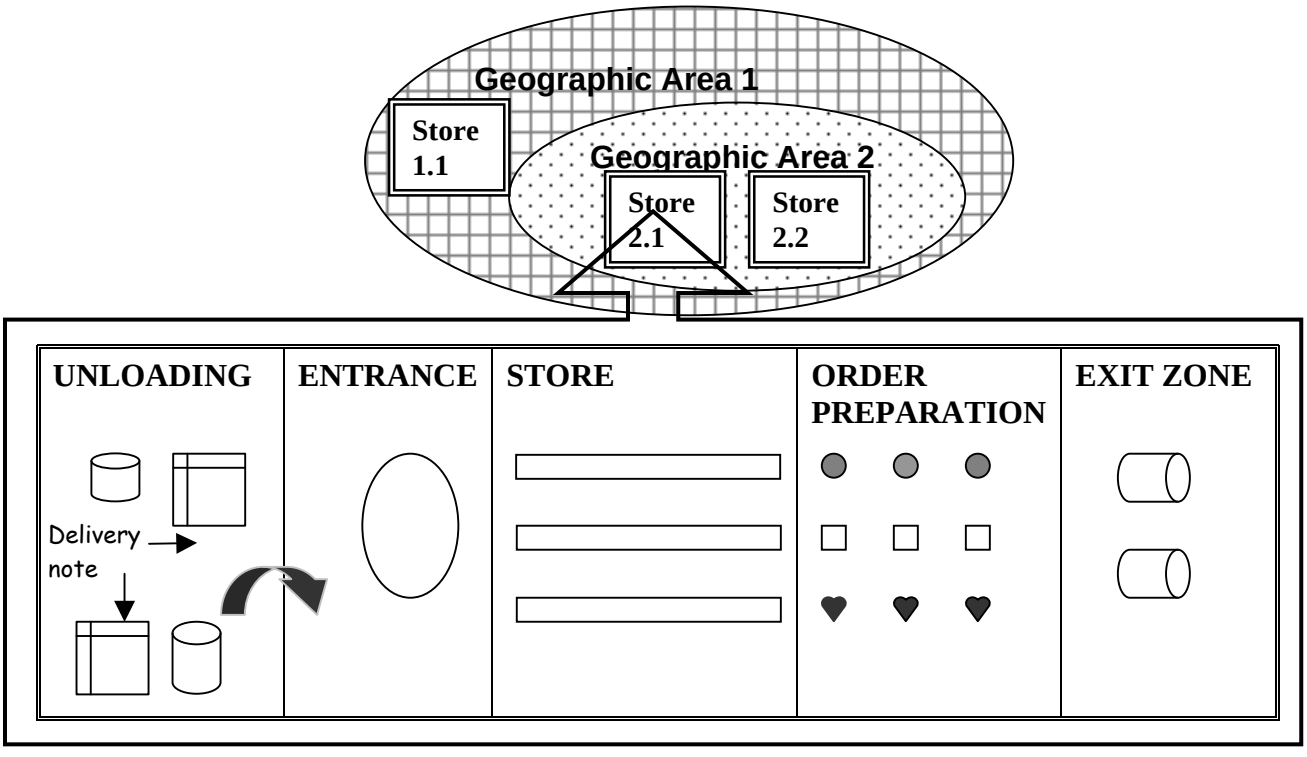


Figure 18: Zones in a store

SOLUTION 1: Composite

- Composite pattern can be used since each area is composed of stores and the stores are composed of zones, but it is not appropriate to use it, because it is not wanted to treat in the same way stores and zones.
- Furthermore, the number of both stores and zones in a store is not usually very large. In a store it is common to have five main zones, with respect to the areas; it depends of how big is the company.

SOLUTION 2: State

- State Pattern is another choice as there are different zones, and in each zone the merchandise usually has a different state.
- But not all the products follows the same route between zones, and they won't be treat in the same way in each zone, so it is not an easy task to find a pattern for all the merchandise in all the zones.

DECISION

Design patterns are not always the best solution. It is true that they are good solutions, because they are probed designs, but sometimes, easier solutions can be used.

In the next figure can be seen the solution chosen:

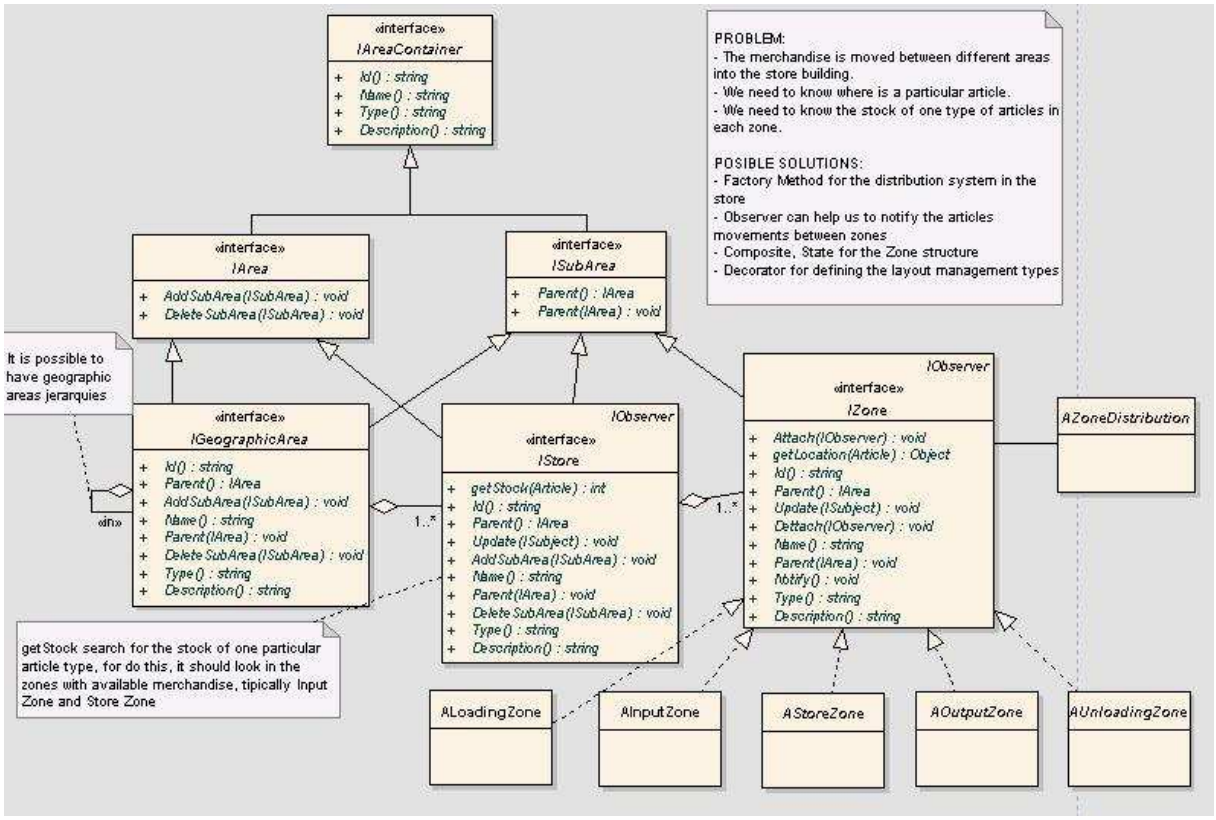

«interface»
IAreaContainer
+ Id() : string
+ Name() : string
+ Type() : string
+ Description() : string
PROBLEM:
- The merchandise is moved between different areas into the store building.
- We need to know where is a particular article.
- We need to know the stock of one type of articles in each zone.
POSIBLE SOLUTIONS:
- Factory Method for the distribution system in the store
- Observer can help us to notify the articles movements between zones
- Composite, State for the Zone structure
- Decorator for defining the layout management types
«interface»
IArea
+ AddSubArea(ISubArea) : void
+ DeleteSubArea(ISubArea) : void
«interface»
ISubArea
+ Parent() : IArea
+ Parent(IArea) : void
It is possible to have geographic areas jerarquies
«interface»
IGeographicArea
+ Id() : string
+ Parent() : IArea
+ AddSubArea(ISubArea) : void
+ Name() : string
+ Parent(IArea) : void
+ DeleteSubArea(ISubArea) : void
+ Type() : string
+ Description() : string
«in»
1..*
IObserver
«interface»
IStore
+ getStock(Article) : int
+ Id() : string
+ Parent() : IArea
+ Update(ISubject) : void
+ AddSubArea(ISubArea) : void
+ Name() : string
+ Parent(IArea) : void
+ DeleteSubArea(ISubArea) : void
+ Type() : string
+ Description() : string
1..*
IObserver
«interface»
IZone
+ Attach(IObserver) : void
+ getLocation(Article) : Object
+ Id() : string
+ Parent() : IArea
+ Update(ISubject) : void
+ Dettach(IObserver) : void
+ Name() : string
+ Parent(IArea) : void
+ Notify() : void
+ Type() : string
+ Description() : string
AZoneDistribution
getStock search for the stock of one particular article type, for do this, it should look in the zones with available merchandise, tipically Input Zone and Store Zone
ALoadingZone
AInputZone
AStoreZone
AOutputZone
AUnloadingZone

Figure 19: Store management

PROBLEM 4.4.2: DISTRIBUTION

This problem is quite complex, so it is going to be split into two smaller problems:

- PROBLEM 4.4.2.1: Equipment problem

Storage systems can be classified depending on the **equipment used in the available space**. The two main classes are with *Corridor* or *without Corridors*, but the user of MSTORE Framework would be able to add more different types of store equipment types. Also, there will be different types of *Locations* depending on the type of equipment. For example, in **Corridor Equipment**, the positions should be located using Corridor, Shelves and Shelf Ids.

In the case of equipment **without Corridors**, more different subgroups can be found:

- **Loose**: The product can be stored on the floor, in heaps, or in big deposits and warehouses.
- **Piled up with blocks**: The merchandise is located in manipulation units, and then they are put in stacks, one close to the following, without any hollow between them. So all the available volume is occupied. A problem can be the weight, so if should be taken care.
- **Rack**: Assembly of a simple structure that will support the load, and it can be assembled and dismantled easily in case of need.
- **Compact with shelves**: Storage using shelves. It is used if the load resistance doesn't support the stocked. Those shelves can be:
  - Dynamic Shelves
  - Drivers

So:

- ✓ There are several zones in the stores and in each zone one different distribution system.
- ✓ Each type of distribution has different way of locate the merchandise, but it is wanted to treat all of them in the same way.
- ✓ Although in the beginning there are only two types of equipment (since all the subgroups of No Corridor Distributions are managed in the same way), it is very possible to have many more (or detail that groups doing more divisions). So it comes up a necessity of making the design extensible better, and with the capacity of support more types.

SOLUTION:

- ➢ Factory Method Design Pattern

It is very useful the ***Factory Method Design Pattern***, that allows us to have a *Creator* (in this case, the *Equipment Factory*), and a *Product* (the *Location*). This product depends on the specific creator in each case.

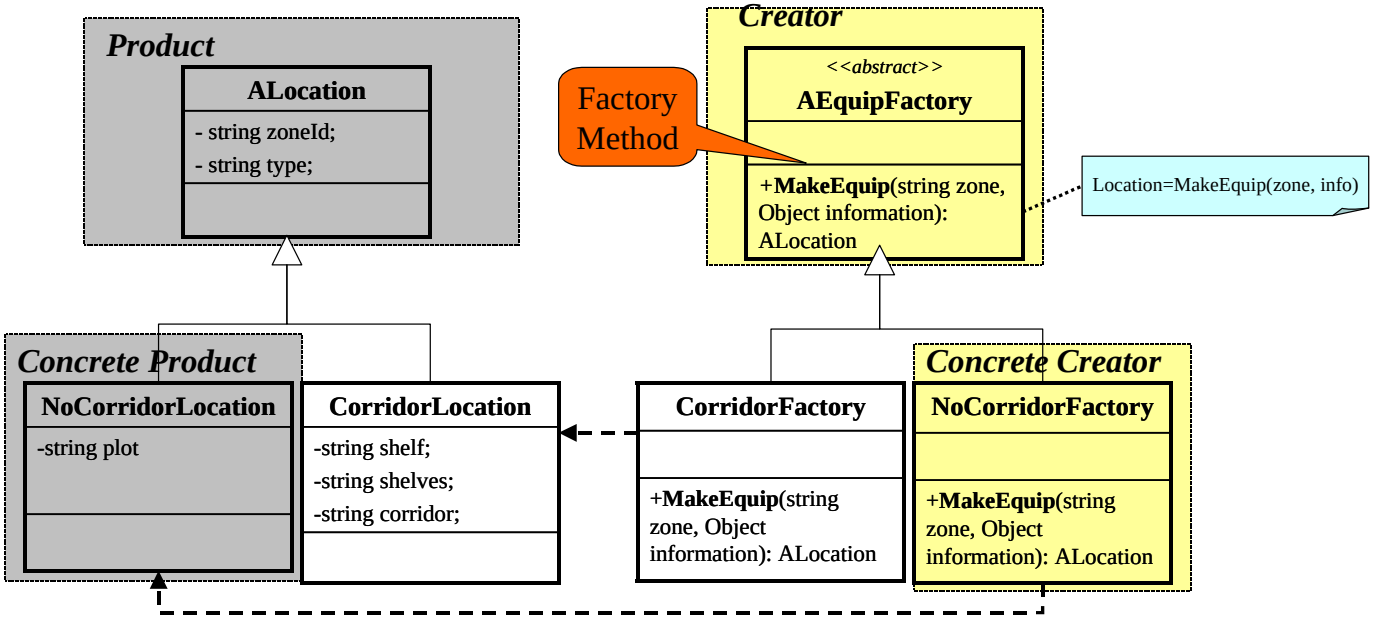


Figure 20: Factory Method Pattern

It is also necessary to point out, that the merchandise, in its way through the store, can change its location several times (as it will be seen in the next problem), and it is very important to know that state, since it isn't useful to have products if they can't be located when it is necessary.

So it should be created an effective method for having all the time all the products located. The basic information it is wanted to obtain could be:

- Geographic Area
- Store
- Zone
- Information to locate the product in the zone. The form of representation depends on the type of equipment that the zone possesses. For example, for the Corridor Equipment, it must be provided the corridor, shelves and shelf for each store unit.

But this information should be given using a standard method, for example a known code (previously fixed) like the following:

```
TtypeofEquipment-GANameoftheGA-SnameoftheStore-ZnameoftheZone-CCorridor-Vshelves-FShelf
```

-PROBLEM 4.4.2.2: Locator problem

Once the equipment distribution of the store is done, it is wanted to establish the possible methods for putting in and taking out the merchandise in the store. It is introduced now the term of *Locator* that is the responsible of locate a free place when a product is getting into the store, and also find products in the store (if the product has to go out from the store, or simply for checking the stock or other information).

The Framework provides the main and more common systems for doing that, but with the use of Design Patterns there can be always added more different ways using the inheritance.

It is going to be seen, dividing them in two main groups (both inherits from the abstract class *ALocator*).

On one hand **the classification for location merchandise**:

- **Input Locators**:
    - *Chaotic Locator*: In this storage system, spaces are assigned when merchandise or products are received without taking into account any determined order. But it is possible to fix some rules for the products location (security reasons, route optimisation, environment conditions…). Almost, the dimensions of the hollows must be appropriate for the products received.
    - *Tidy Locator*: Each product has only a fixed and predetermined place. So the spaces will only keep the products with some particular conditions.
    - The reader can think in other types of input locator as one locator that give us the position in which the manipulation unit fits better.

And on the other hand, **depending on the intern movements**:

- **Output Locators**:
    - *FIFO Locator*: The first product that goes into the store is the first that goes out. It is useful if the store is working with perishable products.
    - *LIFO Locator*: The last products in going into it will be the first going out.

Summarizing, there are the following problems:

- ✓ The locator must create the appropriate location in agreement with the equipment of each zone.
- ✓ It is desirable to have more than one method of merchandise location (depending of factors i.e. if the product is perishable) in each zone.
- ✓ All the locators have similar behaviours, they receive a information about one product (dimensions and type if the locator has to look for a free place, and identification of the product if it is looking for an article type), and they have to reply with the location of the requested store unit, independently on the store unit is a shelf or it is a heap.

SOLUTION:

- ➢ Inheritance: As can be seen in the next diagram, there is an interface *ILocator*, with the common behaviour and methods of all the locators. Then the implementation of the search methods will be in the bottom of the hierarchy.

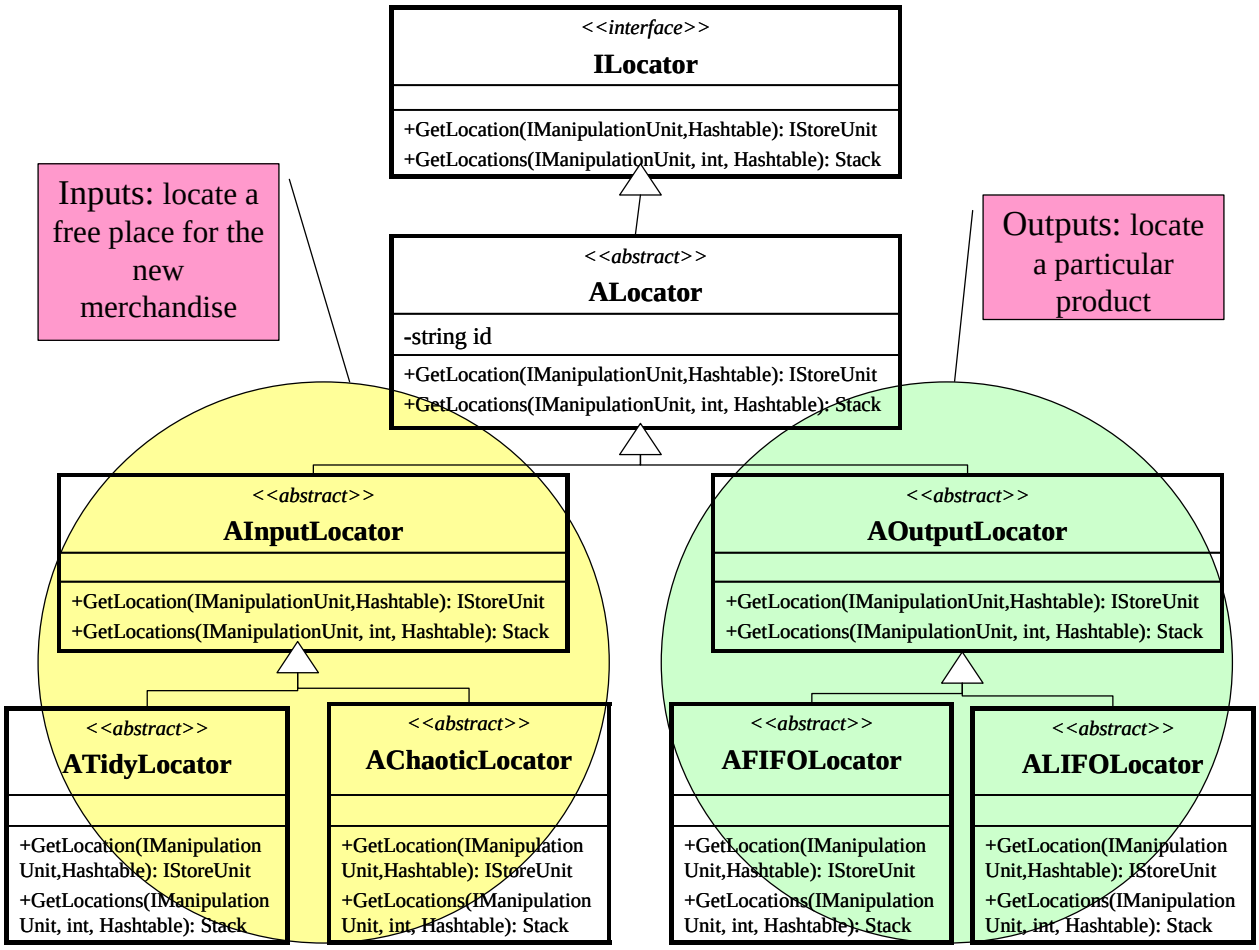
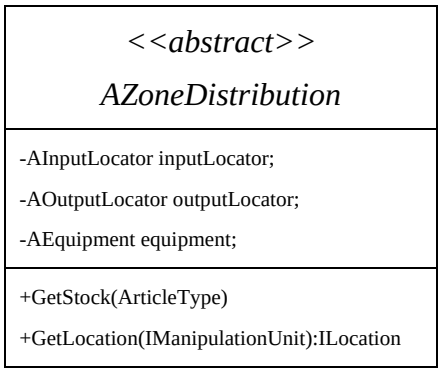


Figure 21: Locators

So, the result of mixing the two solutions is a specific *Zone Distribution* for each zone, and in this zone distribution it should specified the *Input Locator*, the *Output Locator* and the *Equipment* used:



PROBLEM 4.4.3: Physical Distribution

- ✓ It is wanted to offer some support for making diagrams simulating the store for doing easy the management.
- ✓ It is desirable to be able to add more units if the store grows up (a store can change its distribution frequently).
- ✓ All the store units are related with only one figure (depending on the type of store unit, and of course, on the type of equipment used in its zone)
- ✓ Each type of store unit is related with only one feature.

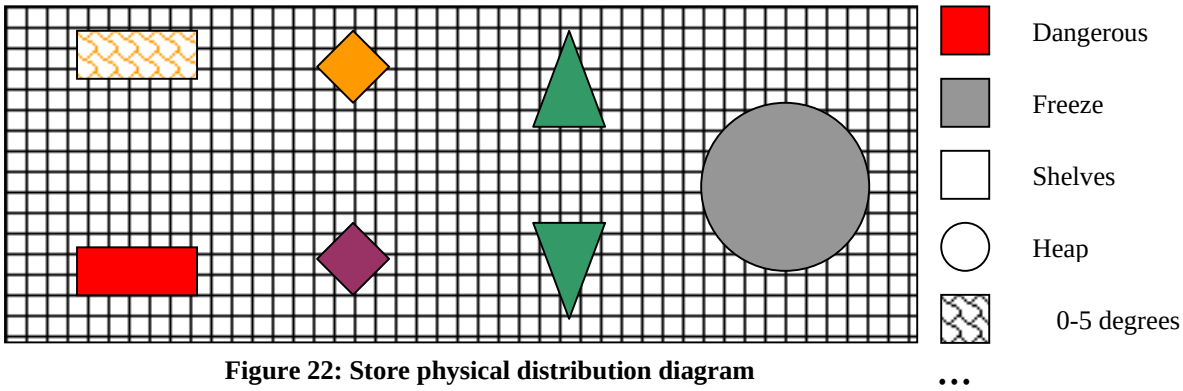


Figure 22: Store physical distribution diagram

SOLUTION 1: Decorator

- ➢ This pattern it is useful if the same unit could have relation with more than one figure, but it is not the case.

SOLUTION 2: Factory Method

- ➢ This is the best solution, as it is known that there are units, with its figure, and types of unit with its feature, and there will be created concrete units and concrete types but there cannot be can't anticipated which subclasses are going to be instantiate.

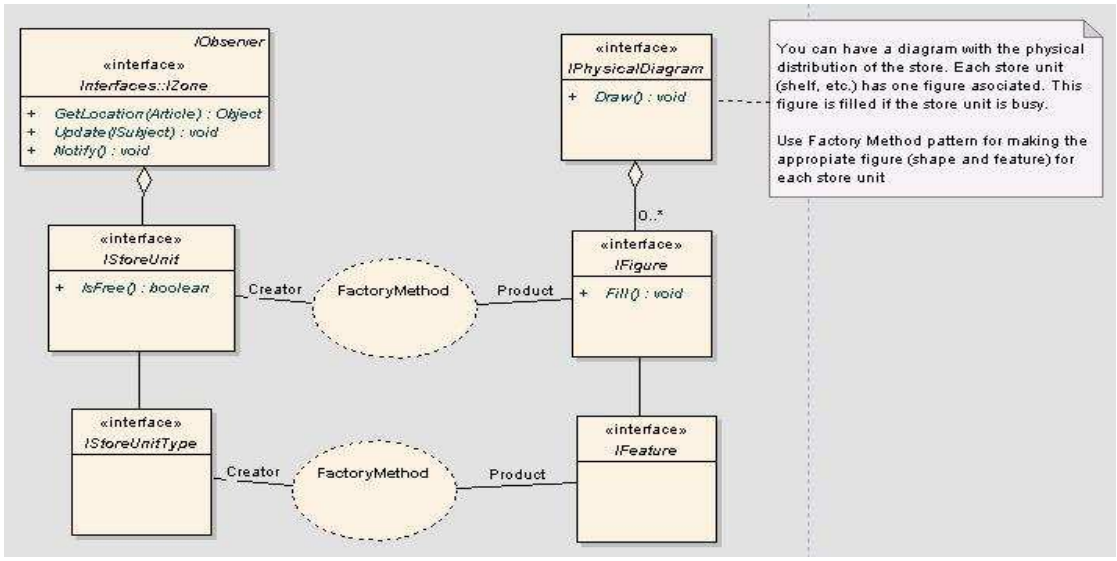


Figure 23: Factory Method pattern solution

4.5 Movements

PROBLEM 4.5.1: KEEPING UPDATED THE STORE INFORMATION WHEN THE MERCHANDISE MOVES

In the following Figure it can be seen an abstract of the previous solution where it was seen an effective method for equipping and locating products in each zone of the store. But the products are moved between these zones and also between stores, so it should be interesting for the company to have them located all the time:

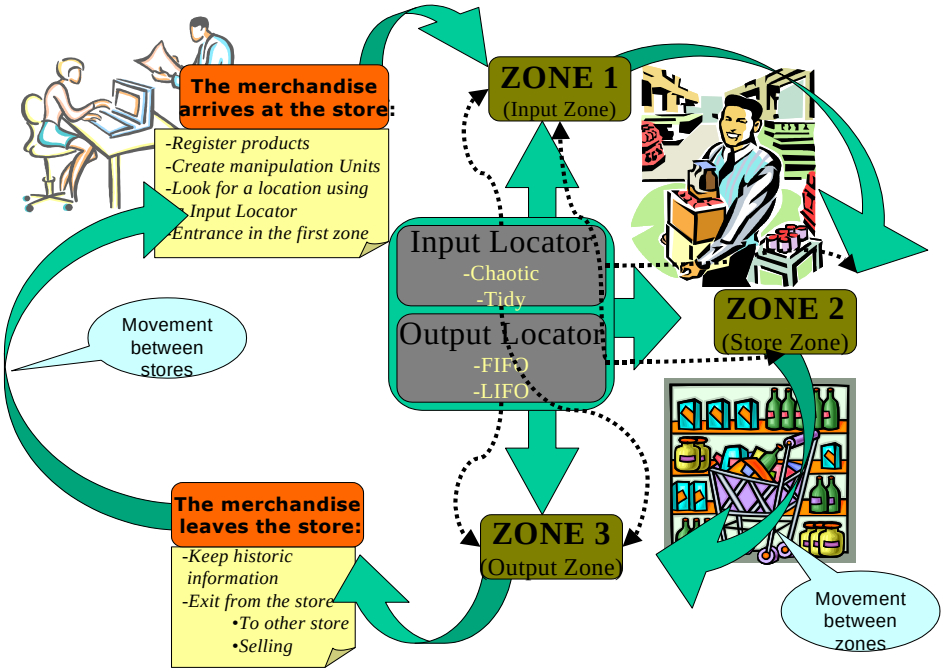


Figure 24: Life cycle of the product in the store

- ✓ In the store there are many merchandise changes
  - Between different areas in the stores
  - Between different stores
- ✓ It is necessary to keep updated the stock of all the articles in each zone and store
- ✓ If it is possible it is wanted to add enlargements with more information to keep

Movements in a store

SOLUTION 1: State

- ➢ As there was pointed before, it was thought in State pattern for dividing the store in zones and in each zone to have one particular state, but the problems for this is that there isn't any pattern for joining merchandise with zones.

SOLUTION 2: Observer

- This is the better solution because it will always have all the data updated, and it can be enlarged as far as it is wanted only inheriting from Observer or Observable.

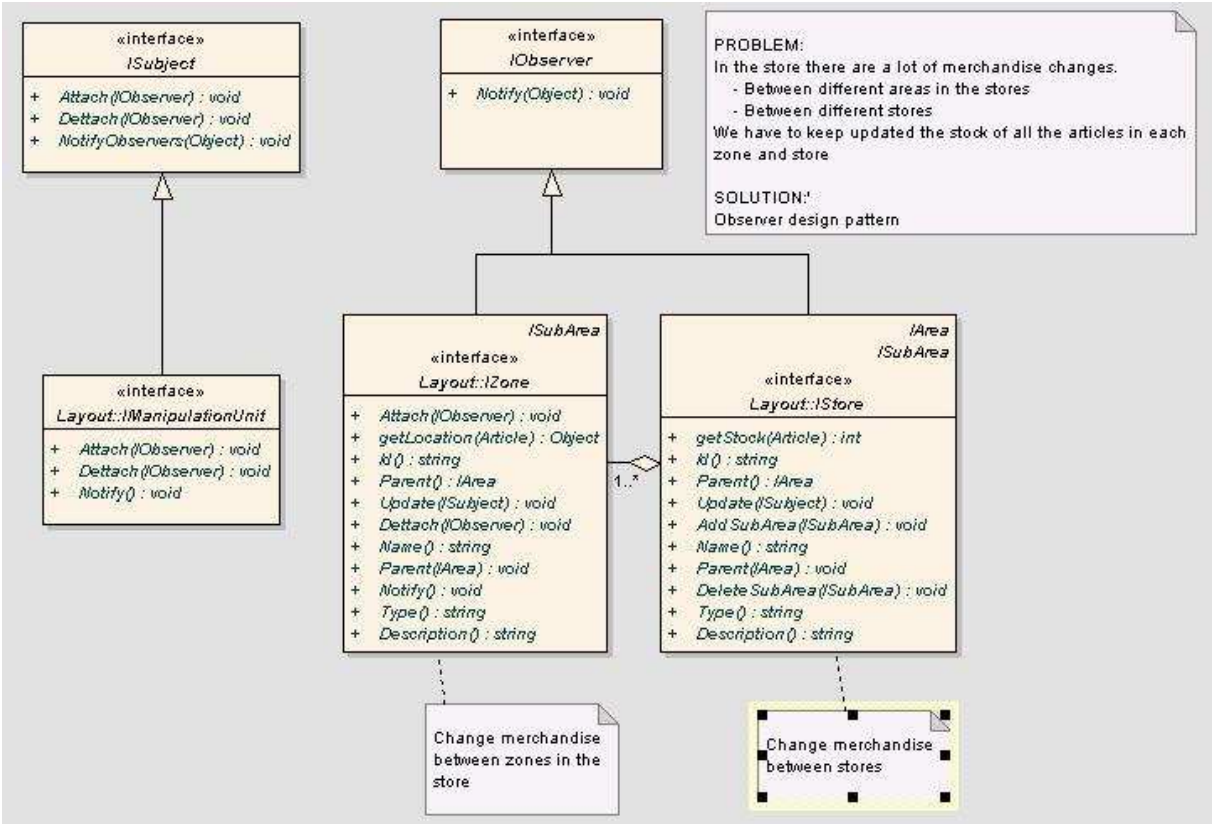


Figure 25: Observer Pattern solution

DECISION: Observer

Finally it was chosen the following diagram for solving this problem.

When the merchandise goes into the store, it is created a manipulation unit for working with it (at the same time, the observers are attached to the manipulation unit). Each time it is wanted to move that manipulation unit, it is called its method move. When it is called, all the observers are informed about the event, and all the information updates.

4.6 Framework Design and Conclusion

Previous chapters explained how were solved the main problems found when developing the framework. After solving those problems, the next step is to put all the chosen solutions together obtaining the definitive framework design. This is not a linear task and it was made in several refinements, but the product achieved is probed to be consistent and usable. The complete framework explanation can be found in the Project Web Page [12]. As explained before is still an alpha version, since it will need feedback from next projects and from future users.

Chapter 5. Applications development using MSTORE Framework

5.1 Common topics in a standard store management application

Two applications have been developed using MSTORE. Although one of their objectives was demonstrating that the Framework could be used for creating different types of store applications, they have also, as almost all the management programs, some common things, which are going to be analysed first. These operations where thought as common for most kinds of store management applications, so they are generic. More problem-oriented operations are seen in next chapters, where there can be seen the specific problems found when developing the pharmacy and the ironmonger.

5.1.1 Analysis

- **ACTORS**

As it is wanted to make the documentation easier to understand, it is going to work with the same types of actors in both applications. In the next table the reader can see the equivalences of each actor in each store, and also their main functions.

ACTOR	GOAL	
	IRONMONGER	PHARMACY
System administrator	Configure parameters Add, modify or remove Areas Start up Shut down	Configure parameters Start up Shut down
Employee	Person that is able to work with the management operations	Person that is able to work with the management operations
Director	Orders Decisions This person can do the operations of the others employees but it is not usual	Orders Decisions This person can do the operations of the others employees but it is not usual
Manager	Resources management Accountancy Process sales Process buying Providers and Clients Management	Resources management Accountancy Process sales Process buying Statistics Providers and Clients Management
Worker	Move merchandise Process inputs Process outputs	Process inputs Process outputs Control sell by dates of the products

Table 1: Actors and Goals

➢ **USE CASES**

Both applications have some common Use Cases. It is going to be followed the notation **Cx** in the common ones (where x is the number of the Use Case):

USE CASE C1	***Manage Company***	
Goal in Context	Actors do the main activities in the Company	
Primary, Secondary Actors	System administrator Employee (Director, Manager, Worker)	
DESCRIPTION	**Step**	**Action**
	1	Manage Resources
	2	Manage Transactions
	3	Manage Merchandise
	4	Manage Store
	5	Manage Users
SUB-VARIATIONS		**Branching Action**
	1-5	User may use Web Services

Table 2: Manage Company Use Case

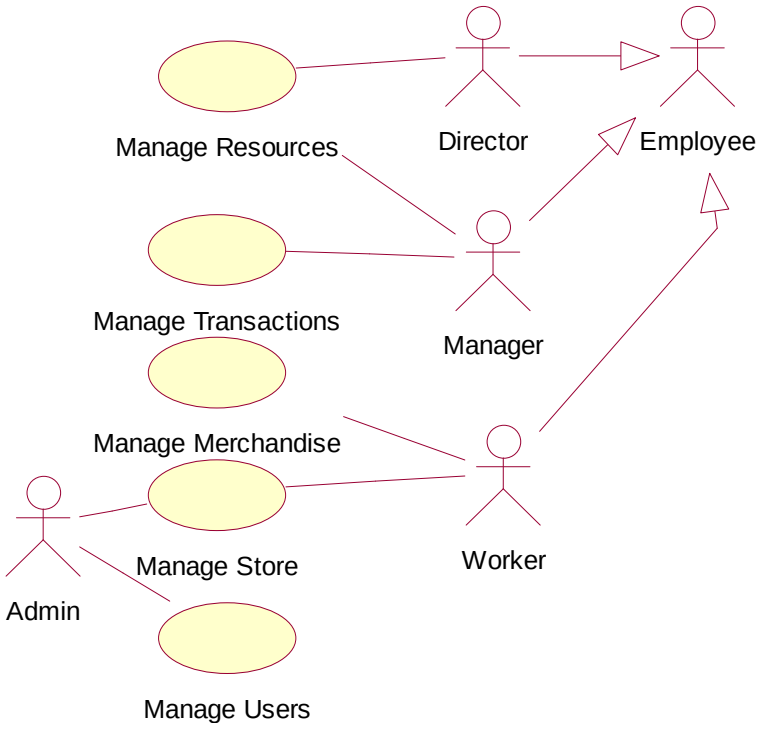


Figure 26: Manage Company Use Case Diagram

USE CASE C1.1	***Manage Resources***	
Goal in Context	Each store manages its own resources. They can be machinery, vehicles, employees, and contacts.	
Preconditions	It is necessary to create, modify, consult or remove a resource The actor can do the action	
Primary, Secondary Actors	Admin Director	
DESCRIPTION	**Step**	**Action**
	1	Choose the type of resource (vehicles, contacts, machinery or employees)
	2	Choose the action it is wanted to do with the resource
	3	Do the action
	4	Close
EXTENSIONS	**Step**	**Branching Action**
	3a	The chosen action is Create: - The user introduce Resource characteristics (For example, if the resource is an employee the data is divided in Personal Details, Contact Details, Economic Data and Labour Data) - The user confirm the action “Add Resource” - The system add the new resource
	3b	The chosen action is Consult Resource information: - The user choose the Resource it is wanted to know more about - The system shows the required information
	3c	The chosen action is Modify Resource - The user choose the resource that is wanted to modify - The user can modify Resource characteristics (For example, if the resource is an employee it is possible to modify Personal Details, Contact Details, Economic Data and Labour Data) - The information is updated
	3d	The chosen action is Delete employee - The user choose the resource that it is going to be deleted - The user confirm the action - The resource is deleted

Table 3: Manage Resources Use Case

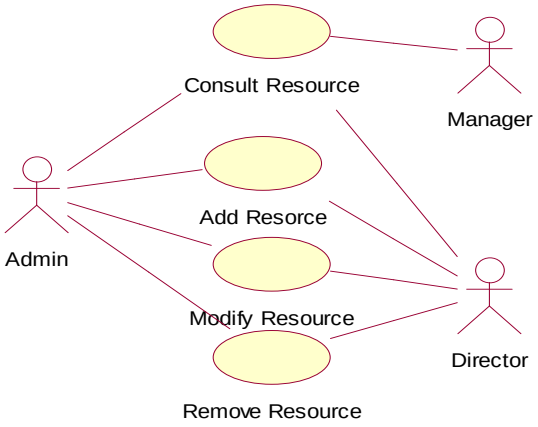


Figure 27: Manage Resources Use Case Diagram

The next Use Case has some common behaviour between the Pharmacy and the Ironmonger that is going to be explained now. Later, in the specific description of each application, this Use Case will be decomposed in other with more particular characteristics.

USE CASE C1.2	***Manage Transactions***	
Goal in Context	A transaction (i.e. Buying, Selling) is performed within a store. The store where is made the transaction is one end; a person (client, provider) represents the other end	
Preconditions	Data about the other end of the transaction is known (i.e. there is knowledge about the client or provider). The store is dealing with some known article types, and the stock of the articles is positive.	
Primary, Secondary Actors	Manager	
DESCRIPTION	**Step**	**Action**
	1	Choose the action it is wanted to do
	2	Do the action
	3	Finish
EXTENSIONS	**Step**	**Branching Action**
	2a	Create new buying - Choose buying characteristics (name, date, etc…). - Choose provider. - Choose store. - Choose articles in the buying (from the provider catalogue), prices, amount, and unit. - Choose payment method.
	2b	View buying information
	2c	Modify buying - Modify buying characteristics (name, date, etc…). - Modify provider. - Modify store. - Modify articles in the buying (from the provider catalogue), prices, amount, and unit. - Modify payment method.

	2d	Delete buying. It must be resolved; if not all of the articles have arrived the buying cannot be deleted yet.
	2e	Create new selling - Choose selling characteristics (name, date, etc…). - Choose client. - Choose store. - Choose articles in the selling, prices, amount, and unit. - Choose payment method. - Manage the stock of all the articles in the selling using the policy attached to the article. If it is necessary, make new a new buying in order to update the stocks.
	2f	View selling information
	2g	Modify selling - Modify selling characteristics (name, date, etc…). - Modify client. - Modify store. - Modify articles in the selling, prices, amount, and unit. - Modify payment method. - Manage the stock of all the new articles in the selling using the policy attached to the article. If it is necessary, make new a new buying in order to update the stocks.
	2e	Delete selling. It must be resolved; if not all of the articles have arrived the selling cannot be deleted yet.
SUB-VARIATIONS		**Branching Action**
	2	One unresolved buying is created automatically when an entry arrives to the shop but not all the expected articles arrive.
	2	One unresolved selling is created automatically when the output is created and there is not stock enough to fill all the output article needing.

Table 4: Manage Transactions Use Case

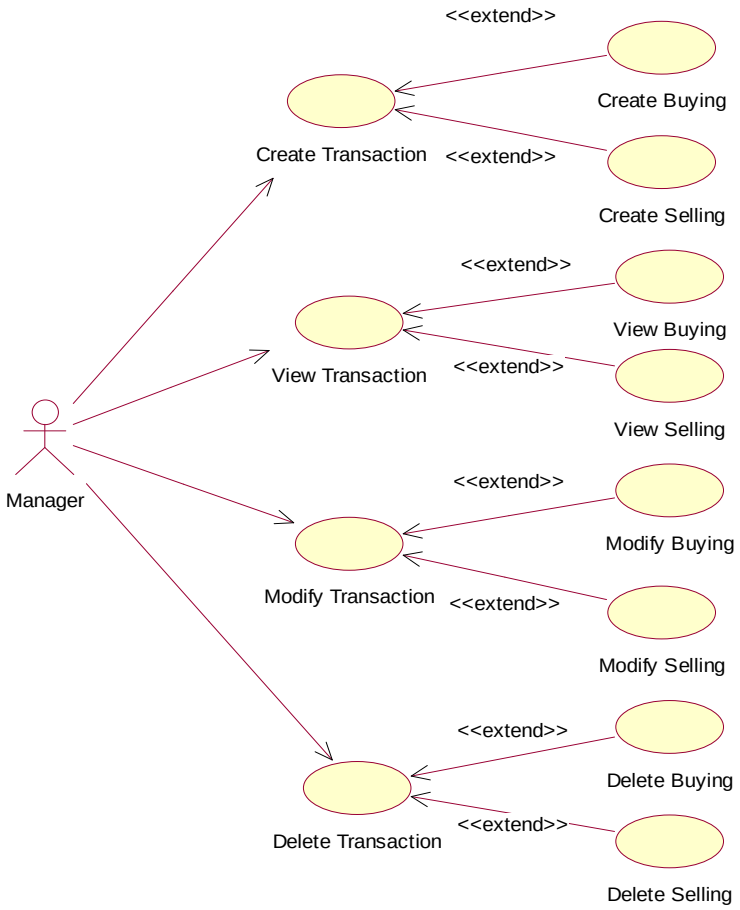


Figure 28: Manage Transactions Use Case Diagram

USE CASE C1.3	***Manage Merchandise***	
Goal in Context	Some merchandise is having interaction with a store, in the entrance of the store (entry) or in the exit (output).	
Preconditions	All the articles, the units and the amounts of the merchandise movement are known. It has been created one transaction that can be attached to the merchandise movement (i.e., one buying can be attached to one entry).	
Primary, Secondary Actors	Worker	
DESCRIPTION	**Step**	**Action**
	1	Choose the action it is wanted to do
	2	Do the action
	3	Finish
EXTENSIONS	**Step**	**Branching Action**
	2a	Register new entry - Choose entry name - Attach buying - Set the store where the entry is registered. - Choose the articles, amounts and units included in the entry. - Create the articles included in the entry

	2b	View entry information - Choose the entry it is wanted to know more about - The system shows the information
	2c	Modify entry - Modify entry name - Modify attached buying - Modify the store were the entry is registered. - Add new articles, amounts and units. - Create the new articles included in the entry - Compare the entry and the buying. Create unresolved buying if necessary.
	2d	Delete entry - Choose entry that is going to be delete - Confirm the action
	2e	Register new output - Choose output characteristics (name, date, comments). - Attach selling - Set the store where the output is registered. - Choose the articles, amounts and units included in the output. - Select the concrete articles included in the output. - Compare the output and the selling. Create unresolved selling if necessary.
	2f	View output information
	2g	Modify output - Modify output characteristics (name, date, comments…). - Modify attached selling - Modify the store were the output is registered. - Add new articles, amounts and units. - Select the new concrete articles included in the output. - Compare the output and the selling. Create unresolved selling if necessary.
	2e	Delete output.
SUB-VARIATIONS		**Branching Action**
	2	When comparing the merchandise movement with the transaction, sometimes they differ in some articles. - When dealing with inputs/buys, if there are articles missing, unresolved buys should be created and registered in order to keep this fact. If some articles are surplus, the user should be able to choose between returning the articles and taking them. - When dealing with outputs/sales, if there are articles missing, unresolved buys should be created and registered in order to keep this fact. If some articles are surplus, the user should be able to return the articles to the store.

Table 5: Manage Merchandise Use Case

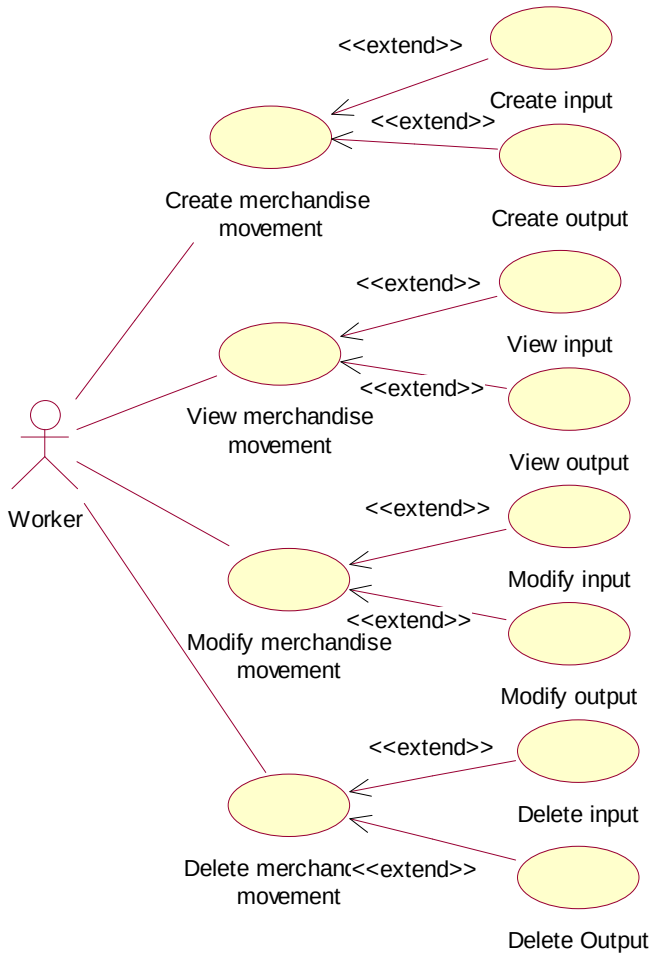


Figure 29: Manage Merchandise Use Case Diagram

It is almost mandatory to have this use case in a store management application; in general, a Company always has one or more store buildings that are divided in zones, or other sub areas. It is possible to abstract the problem and extract the following operations:

USE CASE C1.4	***Manage Store***	
Goal in Context	The store is decomposed in several hierarchical areas that have to be managed	
Preconditions	The administrator must add, modify or delete information about the areas of the company. The actor is authenticated	
Primary, Secondary Actors	System Administrator Director	
DESCRIPTION	**Step**	**Action**
	1	Choose Layout management in the main screen
	2	Choose Area parent of the new sub area
	3	Choose add, consult, modify, remove or equip
	4	Exit the layout management when the configuration is finished

EXTENSIONS	Step	Branching Action
	3a	If the choice is equip a zone: 3a1 Choose zone 3a2 Choose necessary data: - Type of equipment - Data about concrete equipment (corridor, heap…and type of store unit) - Data about volume and weigh 3a3 Accept the new equipment

Table 6: Manage Store Use Case

In the concrete analysis of the real application it will be explained with more details this Use Case:

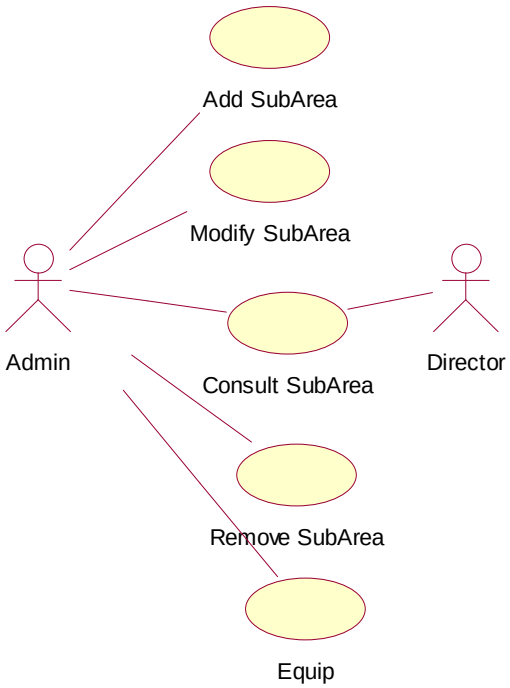


Figure 30: Manage Store Use Case Diagram

As it was fixed in the beginning that the users will be the same for both applications it is going to reuse the Use Case dedicated to manage them too:

USE CASE C1.5	*Manage Users*	
Goal in Context	Different types of users and different operations for each type. So it is important to manage the users that are going to use the application	
Preconditions	The administrator must add, modify, consult or delete information about the users of the company. The application is started up. The actor is authenticated	
Primary, Secondary Actors	System Administrator	
DESCRIPTION	**Step**	**Action**
	1	Log in
	2	Choose action - Add User

		- Consult User information - Modify User information - Remove User
	3	Do the action
	4	Log out
EXTENSIONS	**Step**	**Branching Action**
	1a	If the choice is Login in: Use Case C1.4.1
	3c	If the choice is Add User: - Introduce the data about the new user - Introduce permissions for that user - The system adds the new user
	3d	If the choice is Consult User: - Select user - The system show the information about that user
	3e	If the choice is Modify User: - Select user - Introduce the new data about the selected user - The system updates the information
	3f	If the choice is Remove User: - Select user - The system removes the user
	4	If the choice is Login out: Use Case C1.4.2
SUB-VARIATIONS		**Branching Action**
	1,4	All the actors can do this Use Cases.

Table 7: Manage Users Use Case

USE CASE C1.5.1	***Log in***	
Goal in Context	Different types of users can do different operation, so they must authenticate when they are going to use the application	
Preconditions	The application is started up	
Primary, Secondary Actors	All	
DESCRIPTION	**Step**	**Action**
	1	User introduce login and password
	2	The system validate the login and the password
EXTENSIONS	**Step**	**Branching Action**
	2a	The login and the password are correct - Entrance in the main form - Activate the operations that the user is able to do (depending on the type of user)
	2b	The login and the password aren't correct - Show error - Return to the same window

Table 8: Log in Use Case

USE CASE C1.5.2	***Log out***	
Goal in Context	Different types of users can do different operation, so they must leave the session when they finish	
Preconditions	The application is started up and the session is open	
Primary, Secondary Actors	All	
DESCRIPTION	**Step**	**Action**
	1	User closes the session
	2	The application closes the session safety, keeping all the important information.
EXTENSIONS	**Step**	**Branching Action**

Table 9: Log out Use Case

5.1.2 Design

In this section there is going to be clarified some of the design decisions for the more difficult operations, using sequence diagram for explaining the details of the process. None the entire framework is supposed to be explained in this section, only the operation of some concrete parts that could be interesting for the reader because of the solution chose or the nature of the problem.

For more information, see the complete framework documentation [12]

- **SEQUENCE DIAGRAMS**

SD1 Equipment

First, it will be seen more in depth the equipment operation from the Use Case C1.3 where the administrator of the application must configure all the equipment in each zone. Since each zone can have a different type of equipment (mainly with or without corridors), it is used the Factory Method pattern, as it was suggested in the Chapter 4.

When the administrator begins the Equip process in a zone, the system consult the type of available locations in that zone, previously selected, and depending on that type, it is called the appropriate factory (CorridorFactory or NoCorridorFactory in the Figure 36), and obtained the corresponded location for the data that the administrator had introduced.

With that location it is created a new StoreUnit, which will be aggregated to the zone that is being equipped.

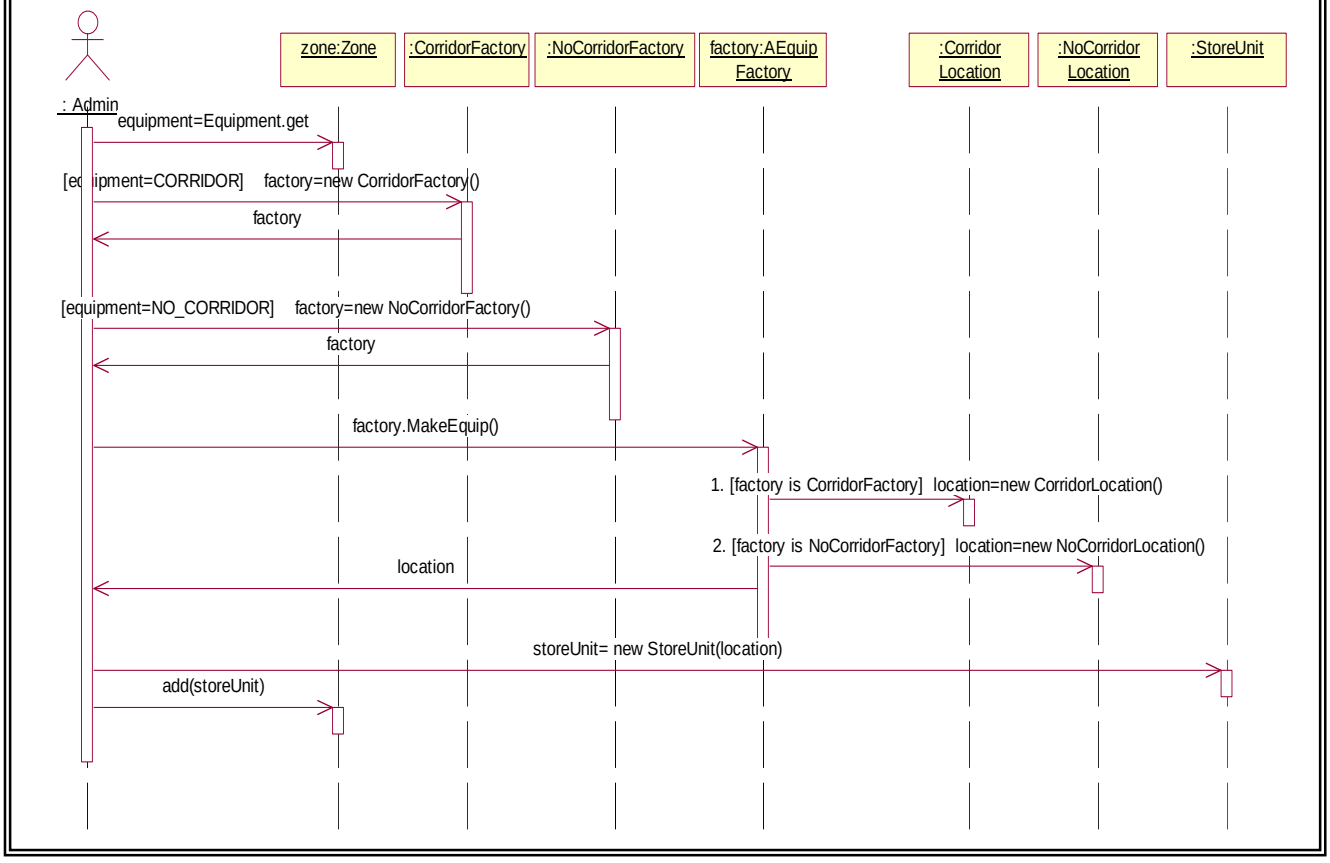


Figure 31: Equipment Sequence Diagram

SD2 New Buying

The shop manager actor performs the process of making a new buying. First the manager introduces the basic data for the buying (name, date, store, payment method and provider). Then, the articles included in the buying must be chosen. The articles are always be part of the provider's catalogue. For each article it is chosen an amount, a price, a unit and a measure. When it is added one article type (observer) to the buying, it is going to be attached to the transaction (that plays a subject role) being included in the observer list.

At the end one document type will be attached to the buying. Depending on the election of the document type, the buying will be initiated with a different builder.

After selecting the different parts of the buying, it is performed with the method DoBuying( ). The method creates the document for the buying and it implements the Notify method: covers the observers list updating the different article types. In this situation these article types will increase their virtual stock with the amount given by the own buying.

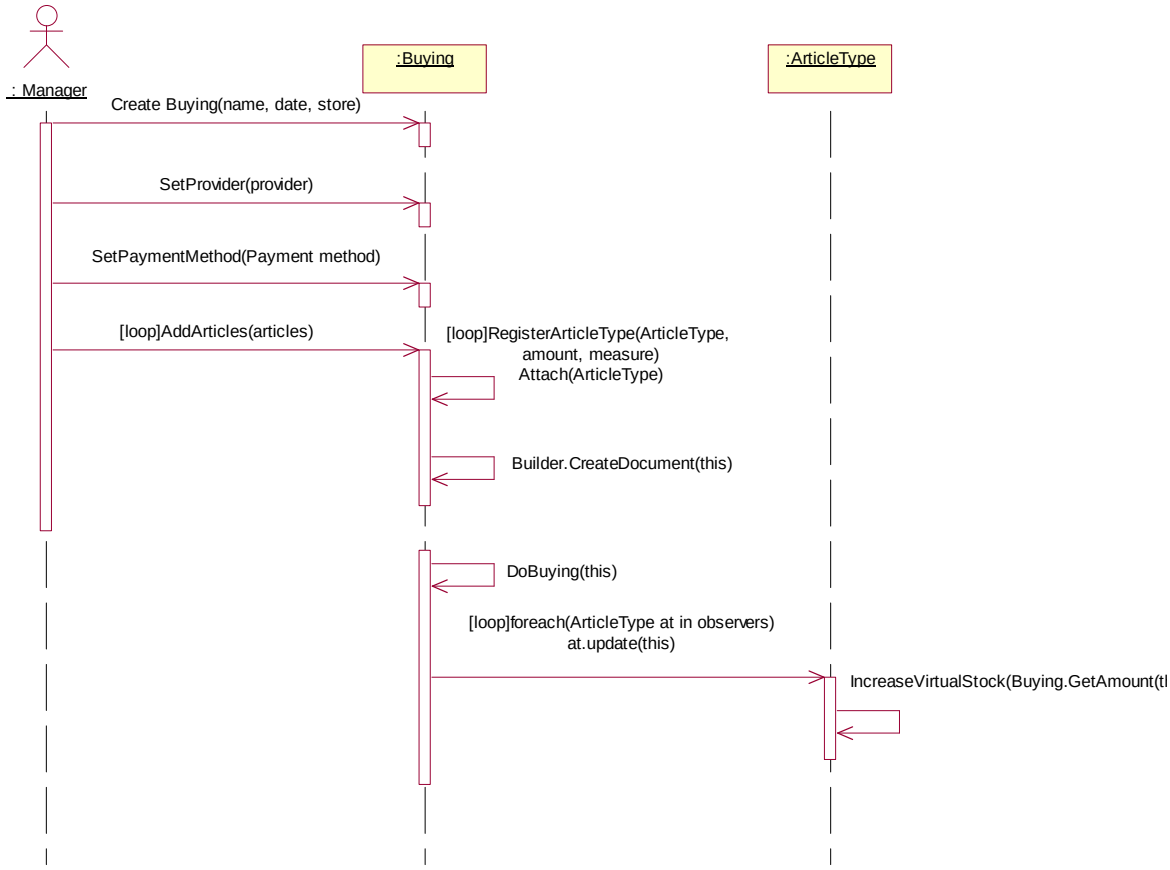


Figure 32: New buying sequence diagram

The Selling process would be almost the same; the observers automatically are going to decrease their virtual stock with the amount given by the selling.

The observer pattern can look like redundant in this example, but this is a key issue in stores management so it is preferred to maintain a complex structure for the stock management.

SD3 New Entry

The next diagram shows the event sequence that take place when a new entry is created. The store worker that receives the entry is the responsible of creating the Entry instance. First all the trivial data (name of the entry, date of the receiving, store where the entry is received…) are added to the entry. Then, one existing buying is attached to the entry. The buying is a logic entity while the entrie represents the physical product entry. Finally, all the article types received are counted and registered into the entry.

When all the information is added, the system proceeds to register the entry in the store. The registration process is the following: for each article type existing in the entry, it is created a number of articles equal to the amount existing in the entry. This amount will be also compared with the one existing in the entry. If there are articles missing, a new unresolved buying will be created using the data of the original buying but the articles already received, and it will be stored in the store. This unresolved buying will be attached to another entry, where there are supposed to arrive the articles missing in the original entry.

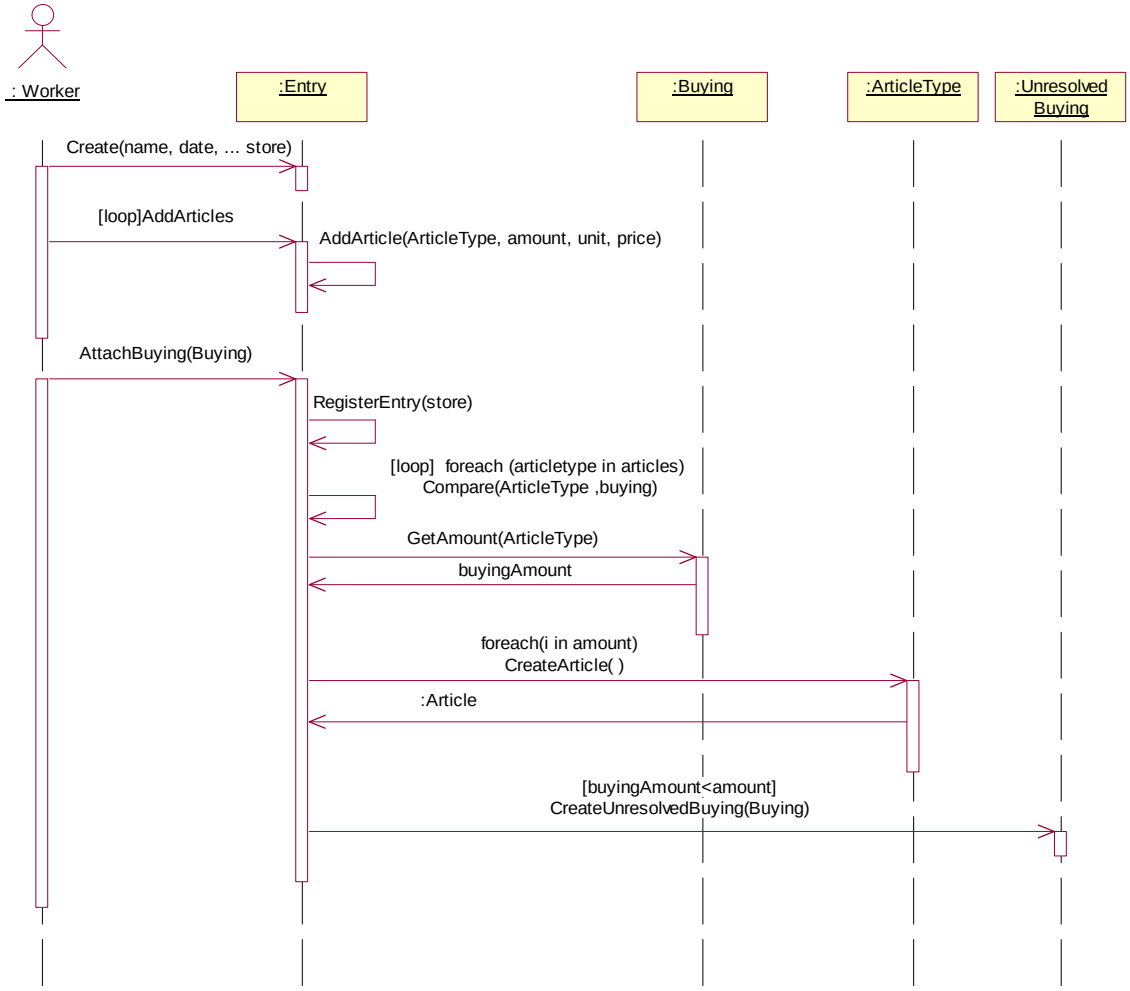


Figure 33: New Entry Sequence Diagram

SD4 New Output

The output process is quite similar to the entry, even while there are some noticeable differences. The first part is the same: there are attached to the output some characteristics such us name, date, and store. After selecting the attached selling, the output is registered. This time there are not going to be articles in the output, as they are took directly from the selling. For all the article types, two steps are going to be done. The first one is managing the stock of the article type using any given stock management policy. The second one is opposite to the one performed within the input: for each article type, it is tried to locate in the shop a number of articles equal to the one desired (we can check it in the selling). If the needing of the article type can be satisfied, the articles with their locations are stored. If there is not stock enough in the store, a new unresolved selling (where there are registered the missing article types) is also created.

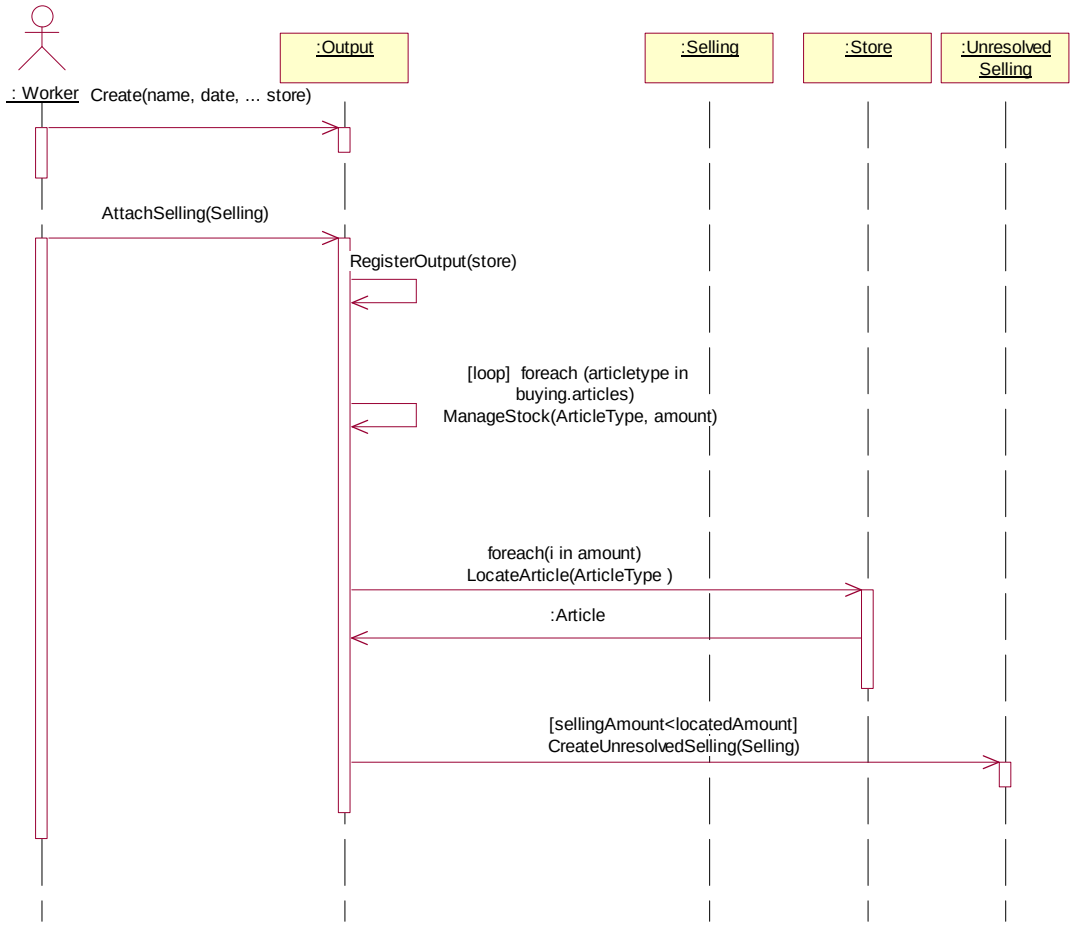


Figure 34: New Output Sequence Diagram

➢ CLASS DIAGRAMS

All the class diagrams can be seen in the project documentation, available at the project Web Page [12]. Here there are only shown the ones with relevant information as an example.

After applying the factory method pattern solution to the problem of the equipment, in both applications, it was obtained a diagram as the next:

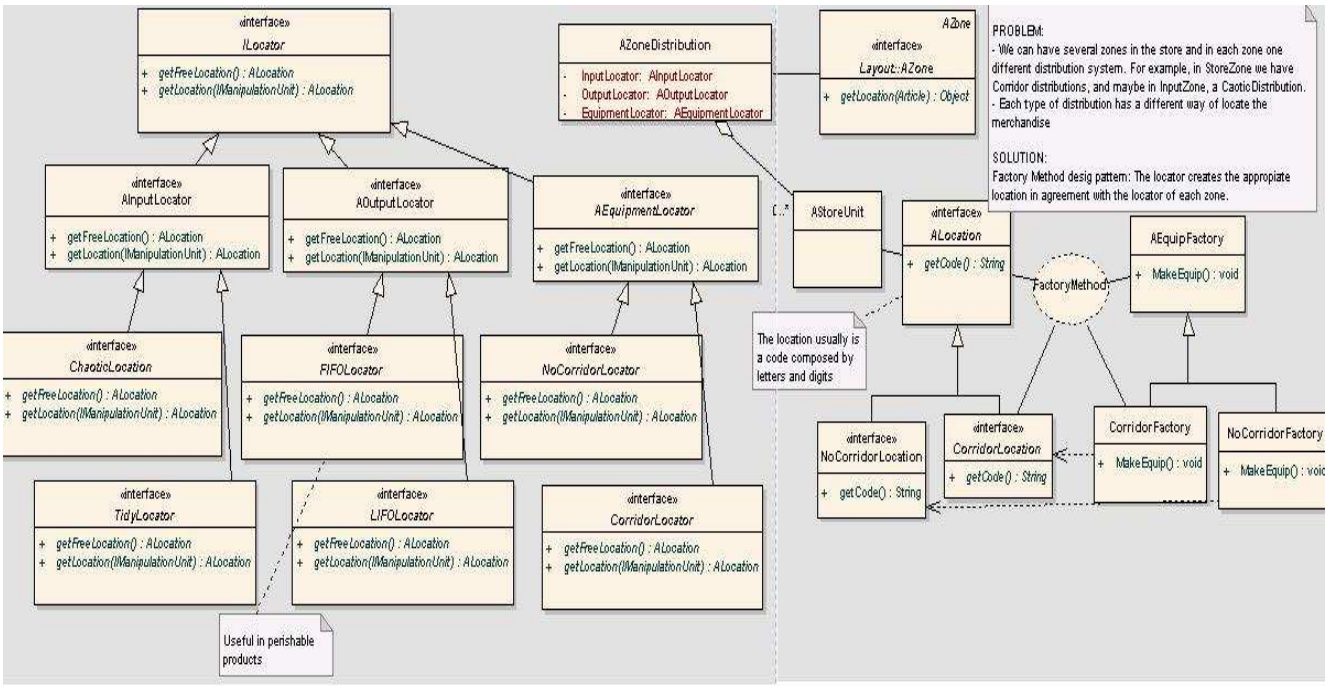


Figure 35: Equipment solution – Factory Method

In this diagram the reader can see that there are two types of equipment (remembering that the applications are only for testing the framework), corridor (with corridors, shelves, shelf…) and no corridor (in chaotic heaps) equipment.

In the diagram is also combined the solution for the Locators problem, without using patterns.

In Figure 36 the reader can see the solution for the transactions applied to the Ironmonger and Pharmacy problem:

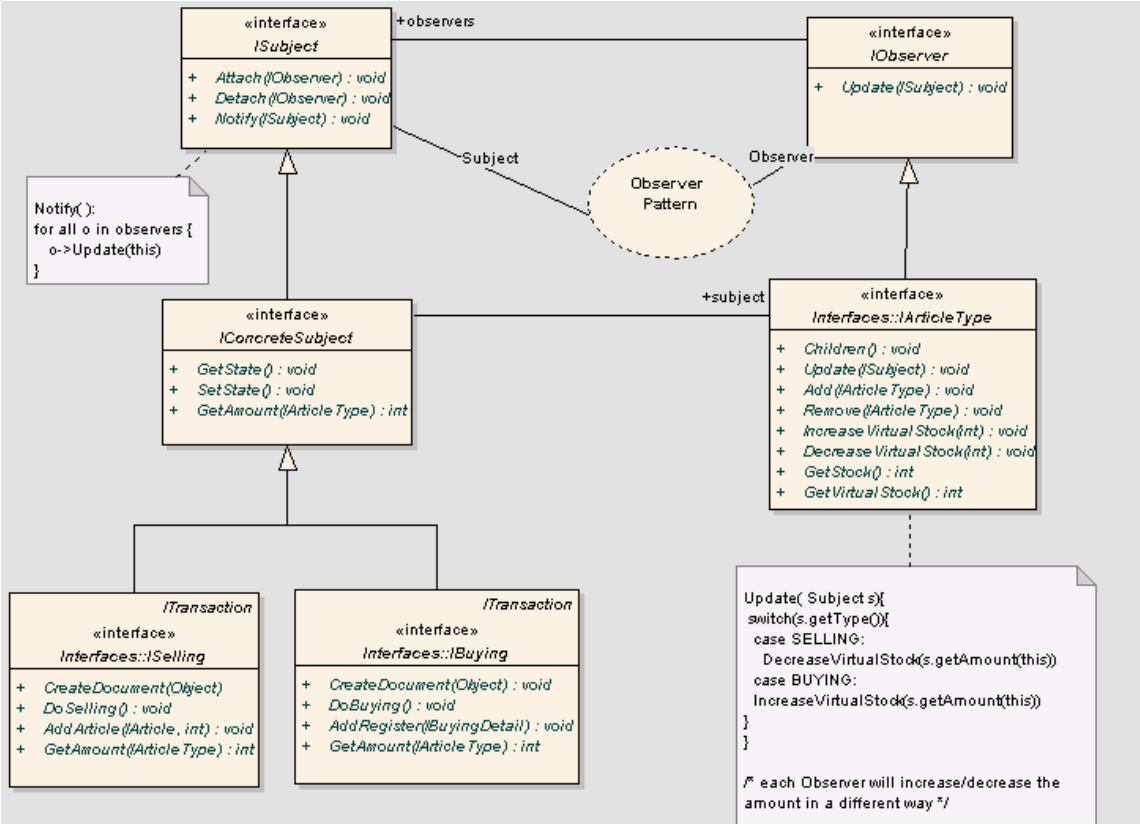


Figure 36: Transactions: Observer pattern solution

The different transactions (buying, selling, entry, output) are going to implement the *Subject* interface, while the article type implements the *Observer* interface. Depending on the transaction performed (for instance make a buying), the observer will change its state in a different way (in the example increase the stock). When making a transaction the observers (that are the products) are just updated, and they update themselves in a different way depending on the transaction. This structure makes the process quite automatic, as there can be notified changes in the stock without making assumptions about who are these objects.

This diagram changes between Pharmacy and Ironmonger but that changes are not interesting here since it will be explained in the sequence diagrams of the specific sections (SD8).

5.1.3 Implementation

- **DESKTOP APPLICATION**

Almost all the management applications have the resources management quite similar; here it is shown the one for the Ironmonger:

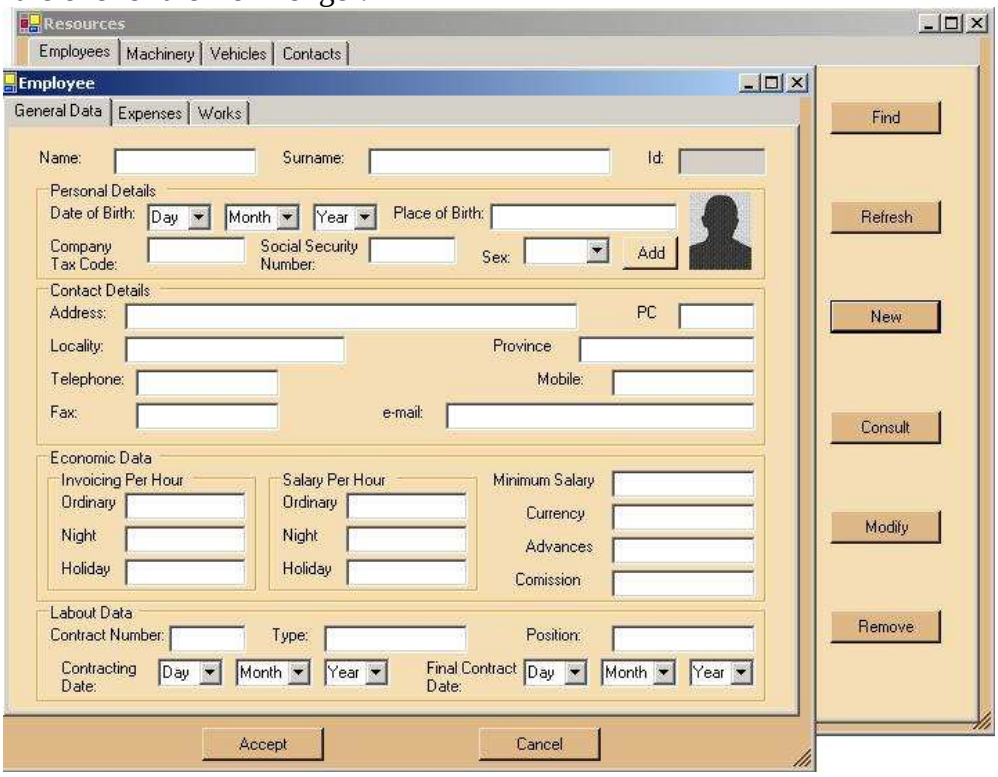


Figure 37: Resources management Screen- Choice: New Employee

Since different users can do different actions it is necessary to log in the system before being able to do any operation:

Figure 38: Log in screen in the Ironmonger

Figure 39: Log in Screen in the Pharmacy

This pair of windows has different interface, but they do the same operation of the Framework.

Some different screens are shown in next sections, since they are quite depending on the specific application.

5.2 Pharmacy: Working with Perishable products

5.2.1 Analysis

➢ **PARTICULAR CHARACTERISTICS OF A PHARMACY'S STORE:**

Next are following the desired characteristics of a pharmacy's store, given from some common pharmacy management systems. Most of them are the common functionalities of any shop management system. Even while the framework gives support for developing almost all the characteristics, by time reasons it was not possible the implementation of a full-functionality pharmacy management system. Next there are explained the common characteristics, and in posterior sections there will be explained with more detail some of the implemented ones.

- **Selling point and client attention.**
  - Fast selling management.
  - Immediate query of client's background, both in a medicament and balance level.
  - Substitutive medicaments management.
  - Automatic prescription type assignment in order to introduce prescription groups with the same type.
  - Reference prices for medicaments. Discounts application depending on special situations.
  - Substitute and generic products location.
  - Unresolved prescriptions management, repayments, and not-paid articles.

- **Cash desk**
  - Register and access to all the movements of the year: inputs and outputs in the cash desk, returned money, pay cash control, credits, cash payment method, credit card payment method.
  - Different reports of cash desk and movements.
  - Control of sales by prescription type.
- **Client operations**
  - Access points to all the sales and balance status of every client.
  - Automatic control for the invoices emission. Possibility of invoice composed by several sales and unique invoices.
  - Creation of different groups of clients (particular clients, companies, etc.).
  - Immediate access to client data.
- **Medicines and other articles**
  - Creation of own article codes.
  - Family creations depending on the user necessities.

- Stock management system per article personalization.
- Creation of bar codes for the articles that need them.
- Inventory level per price and cost.
- Perishable products control.
- Substitutive articles.
- Initial stock establishment.

- **Inventory management, providers**
  - Different order types per provider.
  - Configuration of orders with future providers.
  - Automatic management of the order and optimal unit calculation based in classic methods.
  - Automatic register of benefits per order, register of all the orders.

  - Access from the medicament information to the orders with the medicament, provider, stock levels.

➢ USE CASES

After studying the particular characteristics of the pharmacy, some use cases are added to the existing ones in order to adapt the framework to the particular problems are trying to be solved. The main extensions regard to the fast sales (as the framework is mainly thought for big merchandise movements, not only for small buys and sales) and to the existence of both perishable and not perishable articles. After developing these extensions, they were finally added to the framework, so now the framework gives support for these characteristics. It shows how developing applications using the Framework can help to improve it.

Here are shown the modifications made to the use cases in order to get the new functionalities. These use cases represent variations over the original use case presented in last section, in bold are written the features added to it when developing the new application:

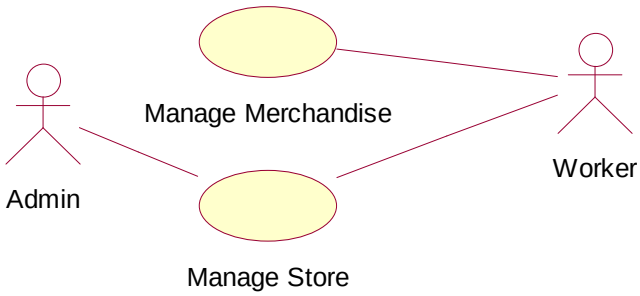


Figure 40: Common Use Cases that were modified in the Pharmacy

USE CASE C1.3	***Pharmacy: Manage Merchandise (Fast sales & perishable articles)***	
EXTENSIONS	**Step**	**Branching Action**

	2a	Register new entry - Choose entry name - Attach buying - Set the store where the entry is registered. - Choose the articles, amounts and units included in the entry. - **Choose expiry date (Perishable + Sell-by date, Not Perishable). For perishable articles, when an output is made, the sell-by-date is checked. If it has expired, the article will be located in a fictitious output (with price 0).** - Create the articles included in the entry - Compare the entry and the buying
	2c	Modify entry - Modify entry characteristics (name, date, comments…). - Modify attached buying - Modify the store were the entry is registered. - **Choose expiry date (Perishable + Sell-by date, Not Perishable).** - Add new articles, amounts and units. - Create the new articles included in the entry - Compare the entry and the buying. Create unresolved buying if necessary.
	2e	Register new output - Choose output characteristics (name, date, comments…). - **Attach selling (optional). If no selling is attached the output is correct.** - Set the store where the output is registered. - Choose the articles, amounts and units included in the output. - Select the concrete articles included in the output. - **If there is attached selling, compare the output and the selling. Create unresolved selling if necessary.** - **If there is not selling attached, there will be created a new resolved selling (with the same articles than the output).**
	2g	Modify output - Modify output characteristics (name, date, comments…). - **Modify attached selling (the choice of a selling is optional).** - Modify the store were the output is registered. - Add new articles, amounts and units. - Select the new concrete articles included in the output. - **If there is attached selling, compare the output and the selling. Create unresolved selling if**

		necessary. - **If there is not selling attached, there will be created a new resolved selling (with the articles added in the modification).**

Table 10: Pharmacy: Manage Merchandise Use Case

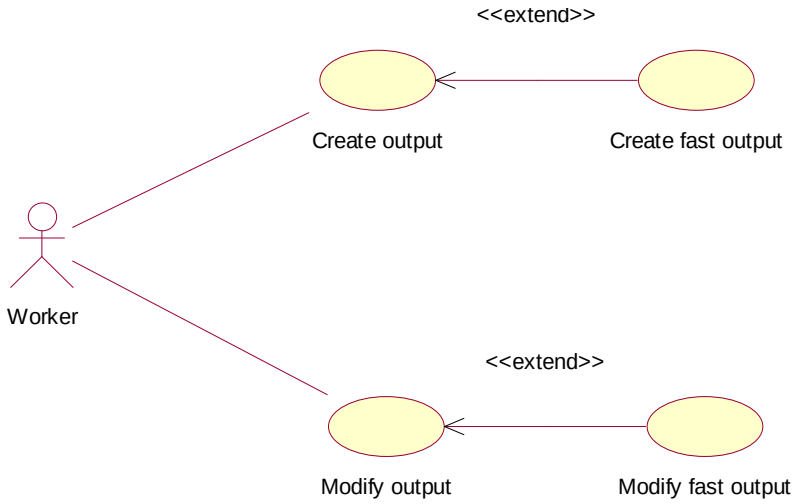


Figure 41: Modified Manage Merchandise Use Case for the Pharmacy

The modifications make much faster the selling process, as it is not necessary the explicit creation of a selling and it is created automatically. It also ensures that the selling and the output will be identical, so there is not need for comparing them and create unresolved buys if something is wrong.

Another important variation regards to the perishable articles. In the original Use Case it was suggested that this variation could be done, but it was not implemented in the framework. The task is quite simple: if the article it is being managed is perishable, a sell-by-date will be added to its characteristics. The output registration process will change also. After the location of one article, if the article is perishable it is going to be checked its sell by date. If it has expired, there will be created a new output (fictitious, with its respective fictitious selling) where the article is located. The price of the article is cero, as it cannot be sold. After the creation of the fictitious output, another article will be searched again.

Here is shown another variation of a use case done for the pharmacy. Again, in bold are marked the new features added to the original use case representing the generic behaviour for all kinds of stores, explained in last section

USE CASE C1.4.1	***Manage Store in the Pharmacy***
Goal in Context	**The store is decomposed in one or more zones** that have to be managed. Each zone has to be equipped

Preconditions	The administrator must add, modify or delete information about the areas of the company. The actor is authenticated	
Primary, Secondary Actors	System administrator Director	
DESCRIPTION	**Step**	**Action**
	1	Choose Layout management in the main screen
	2	**Choose Store parent of the new zone**
	3	**Choose add, consult, modify, remove or equip**
	4	Exit the layout management when the configuration is finished

Table 11: Pharmacy: Manage Store Use Case

Now, the specific Use Cases for this Pharmacy are shown. The used notation is **Px**:

USE CASE P1	***Manage Company***	
Goal in Context	Actors do the main activities in the Company	
Preconditions	The application is ready	
Primary, Secondary Actors	System administrator Employee, Director	
DESCRIPTION	**Step**	**Action**
	1	Manage Statistics
	2	Manage Articles
SUB-VARIATIONS	**Step**	**Branching Action**
	1-2	User may use Web Services

Table 12: Pharmacy: Manage Company

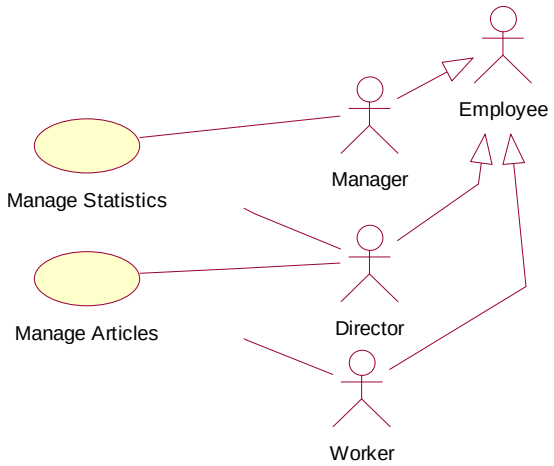


Figure 42: Specific Use Cases Diagram of the Pharmacy

USE CASE P1.1	***Manage Statistics***
Goal in Context	It is wanted to see several statistics about different entities related with the store: benefit per article type, benefit per shop, sales per provider,

	buying per client, etc… These statistics are going to help in strategic decisions.	
Preconditions	There are some instances of the entity wanted to see statistics about. The actor can do the action	
Primary, Secondary Actors	Director Manager	
DESCRIPTION	**Step**	**Action**
	1	Choose the action it is wanted to do
	2	Do the action
	3	Exit
EXTENSIONS	**Step**	**Branching Action**
	2a	The user wants to see statistics about articles.
	2b	The user wants to see statistics about stores.
	2c	The user wants to see statistics about providers.
	2d	The user wants to see statistics about clients.
	2e	The user wants to see statistics about movements
	2f	The user wants to see statistics about employees

Table 13: Pharmacy: Manage Statistics Use Case

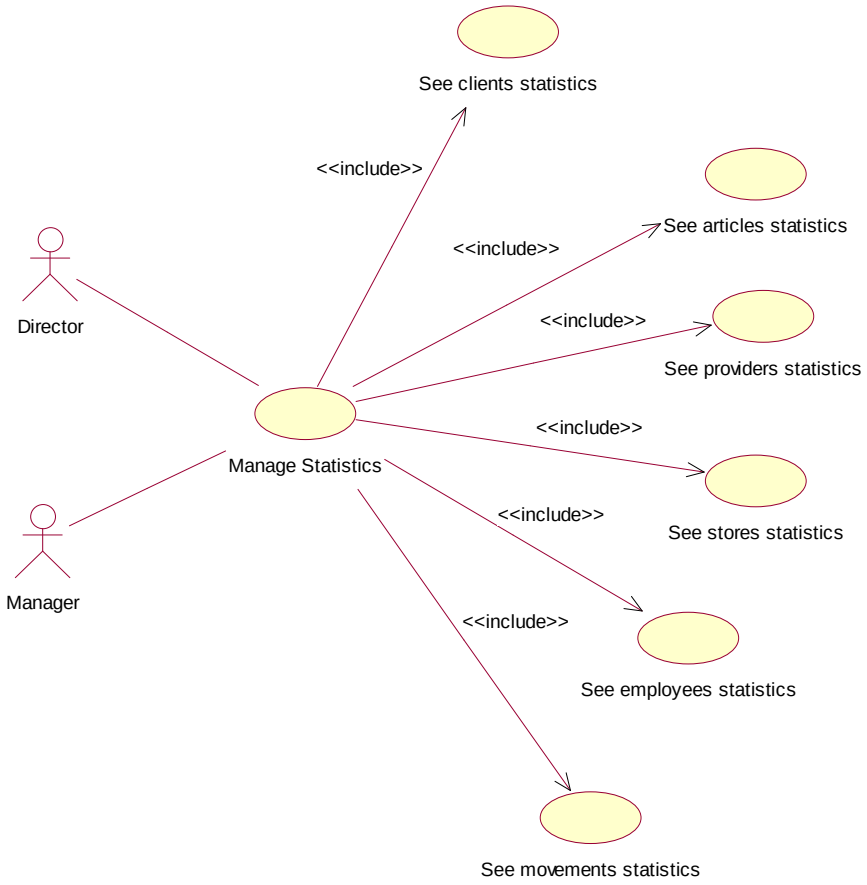


Figure 43: Manage statistics Use Case Diagram

USE CASE P1.2	***Manage Articles***
Goal in Context	Allow different articles types with their concrete characteristics.

Preconditions	It is necessary to create some article types or give more functionality to the existing ones.	
Primary, Secondary Actors	Director Worker	
DESCRIPTION	**Step**	**Action**
	1	Choose the action it is wanted to do
	2	Do the action
	3	Exit
EXTENSIONS	**Step**	**Branching Action**
	2a	Create article type - Choose article characteristics (name, description, comments…). - Choose article volume. - Choose stock management policy. - Select policy parameters (acquisition price, possession price, etc…). - Chose measure used (grams, units, etc…).
	2b	View article type information
	2c	Modify article type - Modify article characteristics (name, description, comments…). - Modify article volume. - Modify stock management policy. - Select policy parameters (acquisition price, possession price, etc…). - Chose measure used (grams, units, etc…).
	2d	Delete article type

Table 14: Pharmacy: Manage Articles Use Case

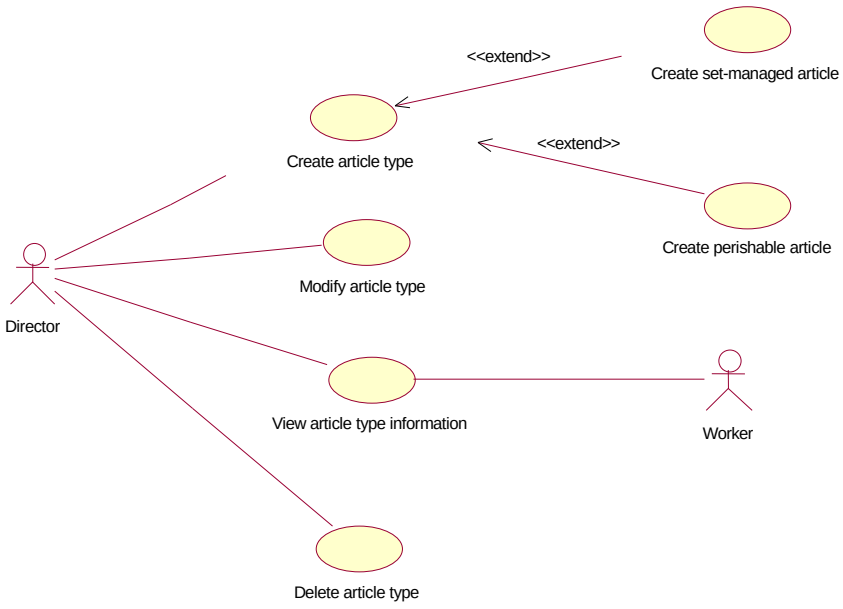


Figure 44: Manage Articles Use Case Diagram

5.2.2 Design

➢ **SEQUENCE DIAGRAMS**

The next sequence diagrams illustrate the use cases presented in last section. The explanation can be found within the particular use cases:

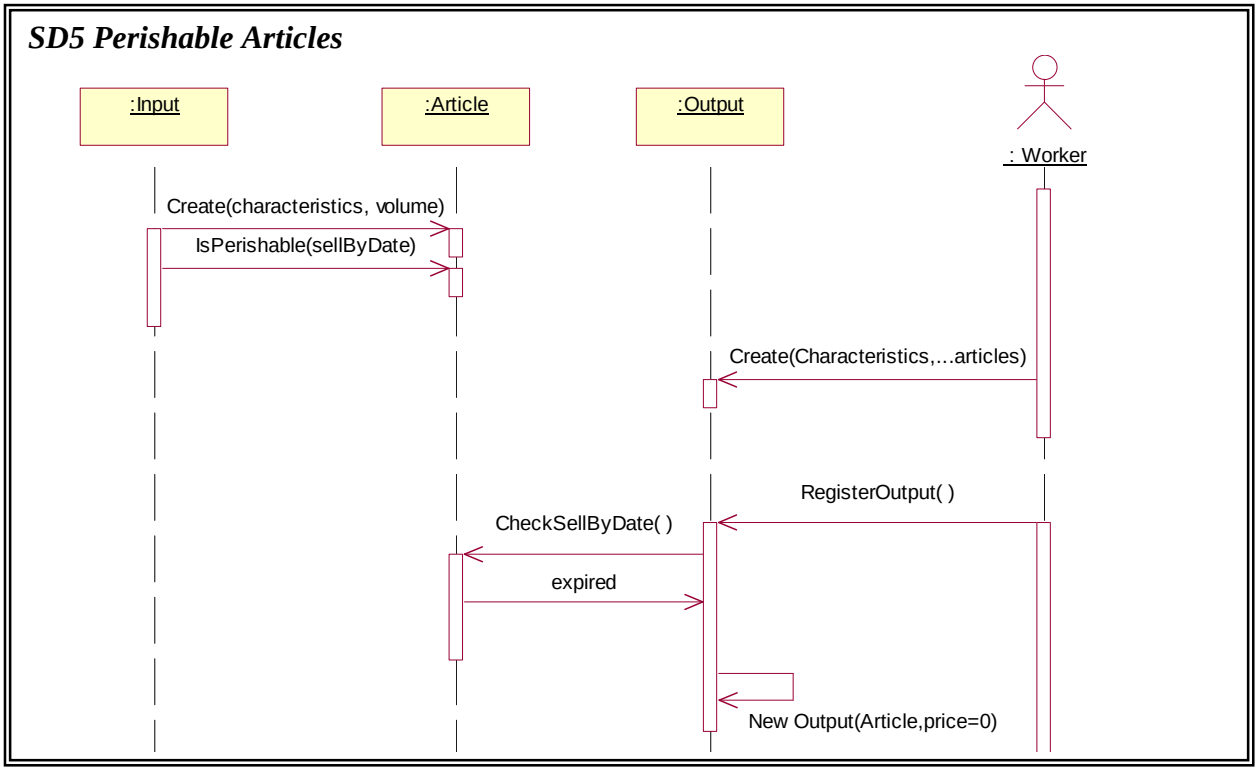


Figure 45: Create Perishable Article Sequence Diagram

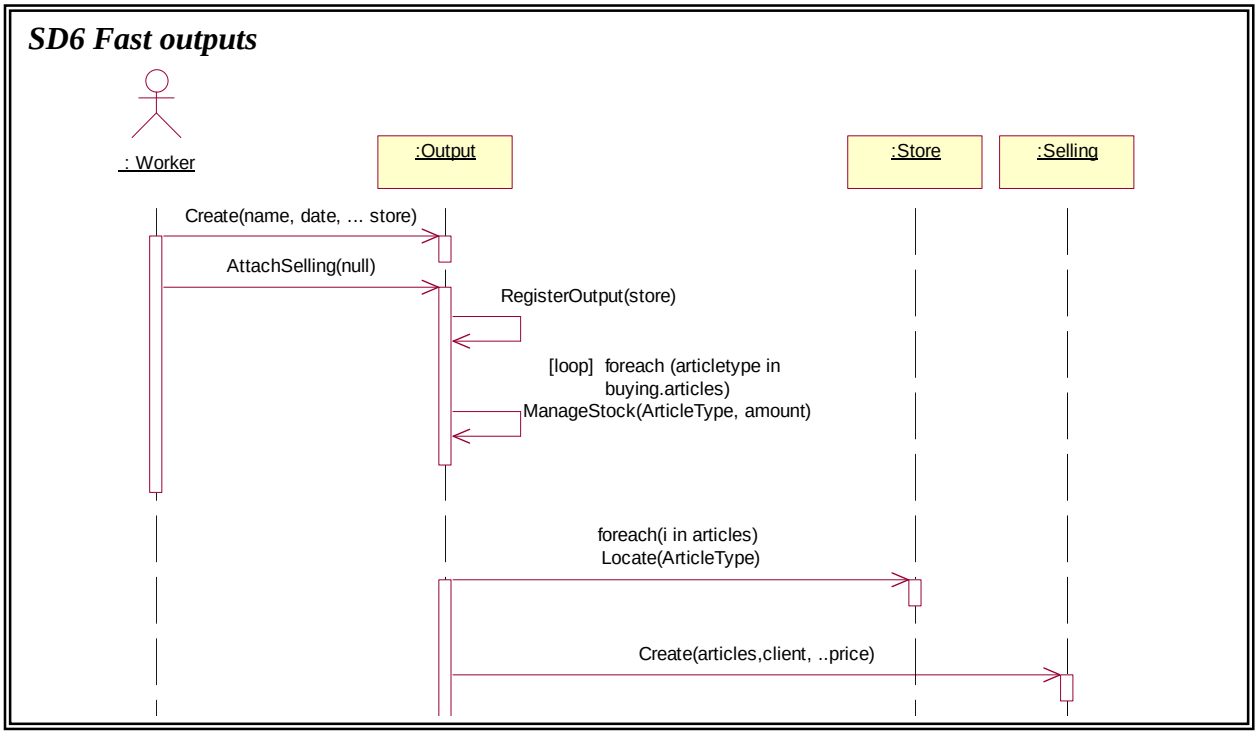


Figure 46: Fast Output Sequence Diagram

5.2.3 Implementation

➢ **DESKTOP APPLICATION**

The object-oriented design of the framework, using design patterns and 3 layers (interfaces, abstract, and concrete classes) make easy and fast the pharmacy management program development. Anyway, as it was the first program (as well as the ironmonger's program) developed using the framework, many of the development time was used debugging the framework and adding new functionalities to it. Little code was necessary only for the pharmacy program, and most of it was related with the user interface and the presentation. Some of the main forms can be seen here, for further information consult the project web page [12].

Figure 47: Main Screen of the Pharmacy Management application

This is the main screen of the Pharmacy. As the reader can see it is necessary to login for being able to do the rest of operations. When the user finishes the work it should log out.

These operations are related with the database, as they convert the database tables in objects that can be used within the application.

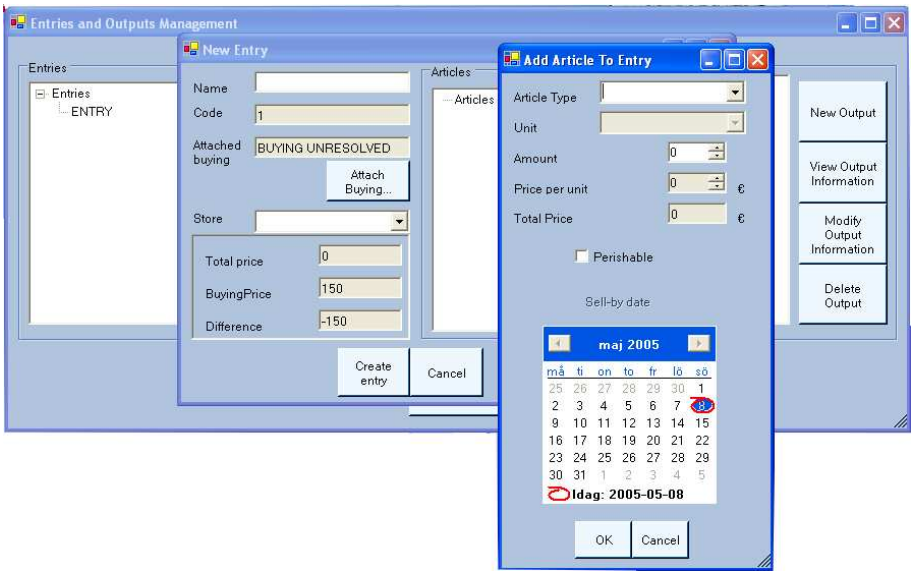


Figure 48: Add article to entry screen

Last window shows how the worker can add an article to an entry. This task is done each time the store receives merchandise. With this screen the information introduction is manual, but in a real store it is usual to have bar codes for making this easier.

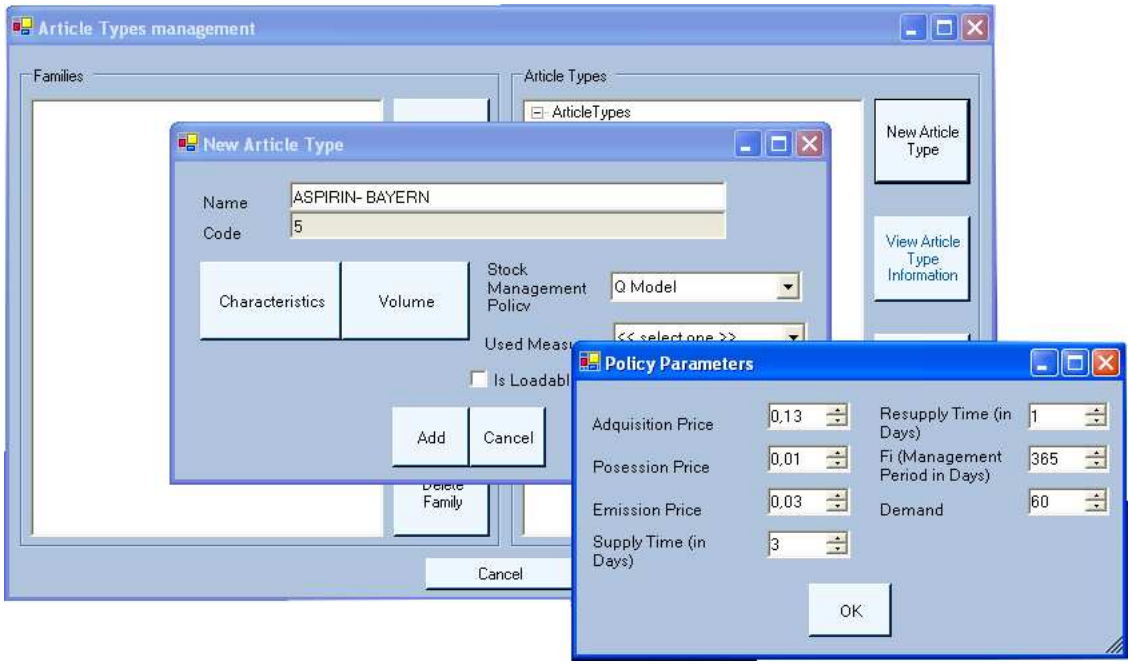


Figure 49: Set Stock Management Policy parameters screen

Some parameters must be fixed. In Figure 53 it is shown the form for filling them. This task is done when an article type is created or modified.

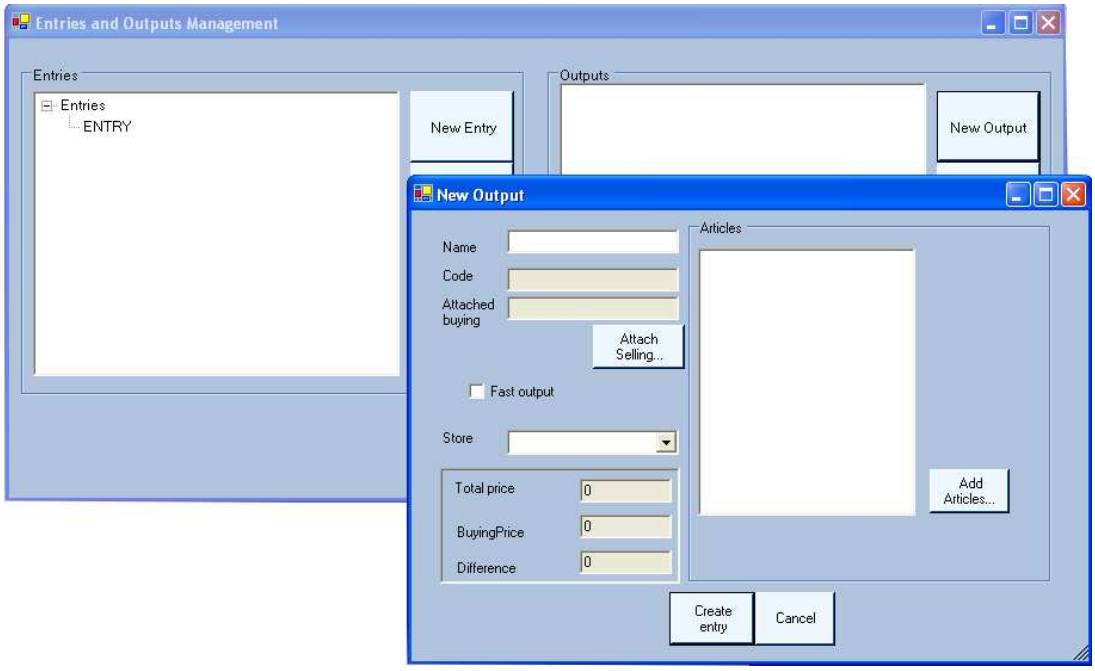


Figure 50: New Output screen

Finally it is wanted to show how create an output in the Pharmacy. As the reader can see in the window, it is necessary to fill some information and the articles that compose the output, that are selected from a list.

Some of the screens show operations that can be similar to any other store management application (such us the main screen or the set stock management policy screen), while others show operations more specific to our application (such us the sell-by date for the products of the entry or the fast output).

➢ WEB SERVICES

Some of the application services have been made public over the Internet in form of Web Services. Any client using a web browser can invoke these methods, using some given parameters and the client can check the results that are returned as a SOAP message. For further information about web services, see the Used Technologies chapter and the references [16].

No consumer was created for these web services, only a producer. The user can also notice that only a small set of services (from the big variety that can be found in the framework) are offered as web services. The reason is that these services were only implemented as a first step, and they are not supposed to offer full-functionality yet. Next versions of the framework will use a more complete set of services, as well as a consumer program.

Here can be seen the browser's appearance when invoking our application via Internet. The application includes a small amount of data loaded in order to show some examples. There is also shown the invocation of a method, with the corresponding results.

This browser capture shows the main window of the web application. Small descriptions of the application, as well as the operations are shown. As can be seen, no effort was made in the graphic design, as here it was only tried to make a first iteration with the web services in the framework.

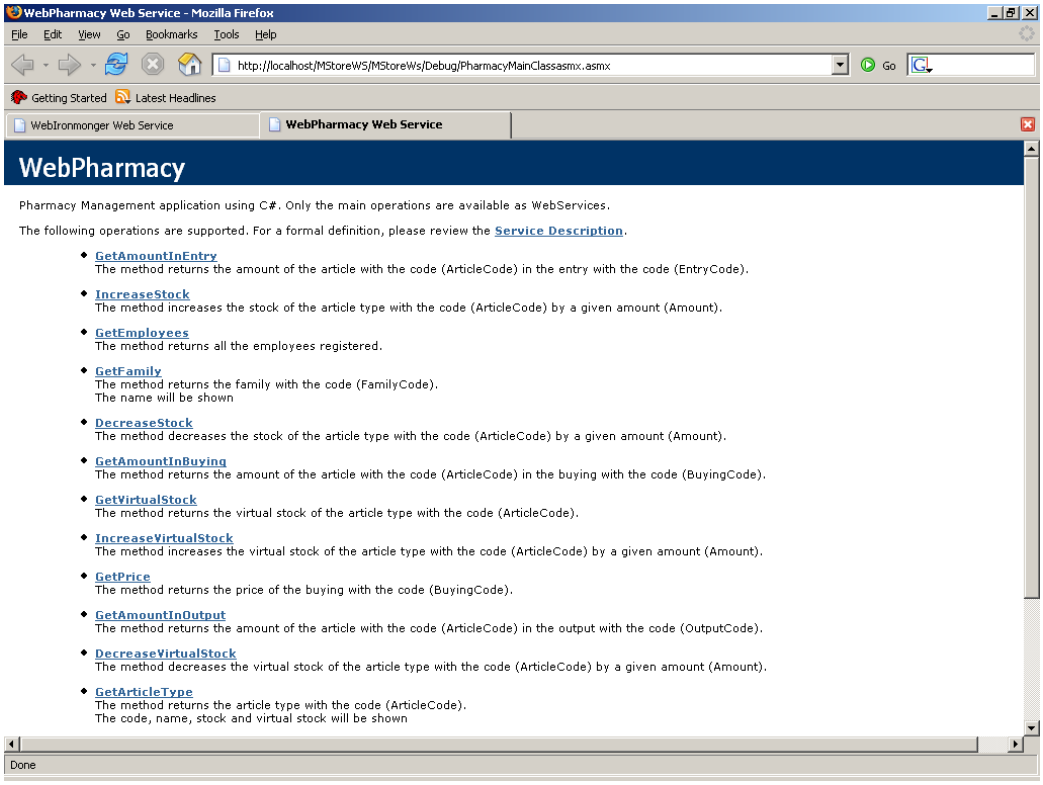

WebPharmacy Web Service - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/MStoreWS/MStoreWs/Debug/PharmacyMainClassasmx.asmx
Getting Started Latest Headlines
WebIronmonger Web Service
WebPharmacy Web Service
WebPharmacy
Pharmacy Management application using C#. Only the main operations are available as WebServices.
The following operations are supported. For a formal definition, please review the Service Description.
GetAmountInEntry
The method returns the amount of the article with the code (ArticleCode) in the entry with the code (EntryCode).
IncreaseStock
The method increases the stock of the article type with the code (ArticleCode) by a given amount (Amount).
GetEmployees
The method returns all the employees registered.
GetFamily
The method returns the family with the code (FamilyCode).
The name will be shown
DecreaseStock
The method decreases the stock of the article type with the code (ArticleCode) by a given amount (Amount).
GetAmountInBuying
The method returns the amount of the article with the code (ArticleCode) in the buying with the code (BuyingCode).
GetVirtualStock
The method returns the virtual stock of the article type with the code (ArticleCode).
IncreaseVirtualStock
The method increases the virtual stock of the article type with the code (ArticleCode) by a given amount (Amount).
GetPrice
The method returns the price of the buying with the code (BuyingCode).
GetAmountInOutput
The method returns the amount of the article with the code (ArticleCode) in the output with the code (OutputCode).
DecreaseVirtualStock
The method decreases the virtual stock of the article type with the code (ArticleCode) by a given amount (Amount).
GetArticleType
The method returns the article type with the code (ArticleCode).
The code, name, stock and virtual stock will be shown
Done

Figure 51: Pharmacy Web Services, mainframe

Any method can be invoked using the correct parameters. The result, as can be seen in next figure, will be returned as a SOAP message. This example only asks for a collection of the article types existing in the store, showing its stock and virtual stock. In the future, more complex examples will be developed, but they are out of the scope of this project.

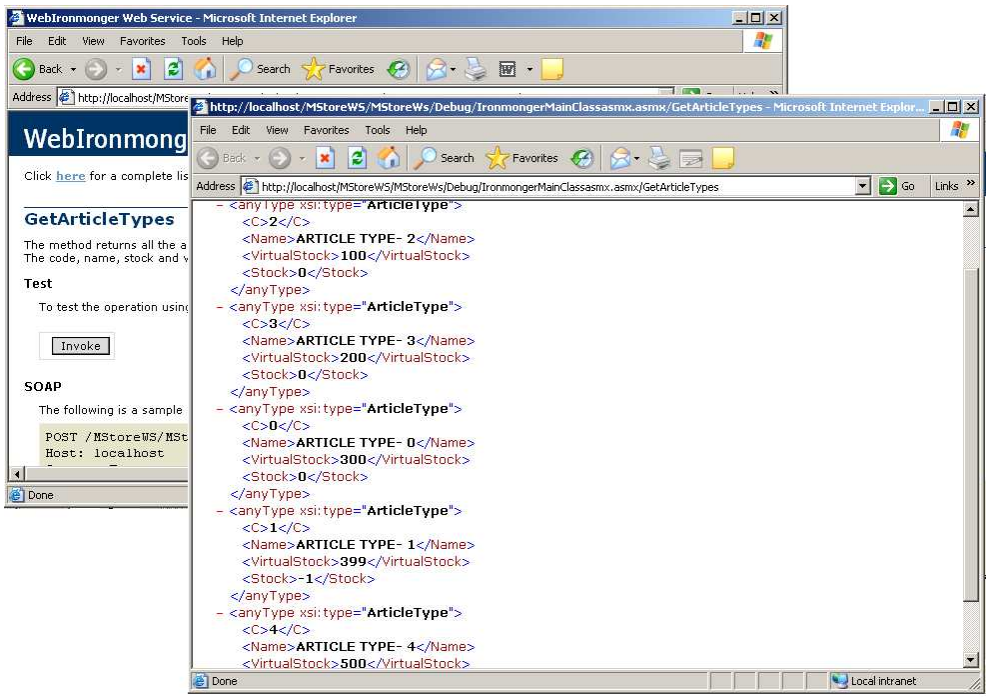


Figure 52: Invocation of one method and result

Chapter 6. Used technologies

6.1 .NET Framework

6.1.1 Basic .Net platform architecture: Framework Description & Main Components

The .Net Framework [18] [19] is an infrastructure where there is collected a set of languages and services that simplify hugely the applications development. By means of this tool it is offered a highly distributed programming **environment** that allows the creation ship of solid and easy-to-extend applications. The main components of this environment are:

- Compilation **languages**.
- . Net Classes **Library**.
- **CLR** (Common Language Runtime).

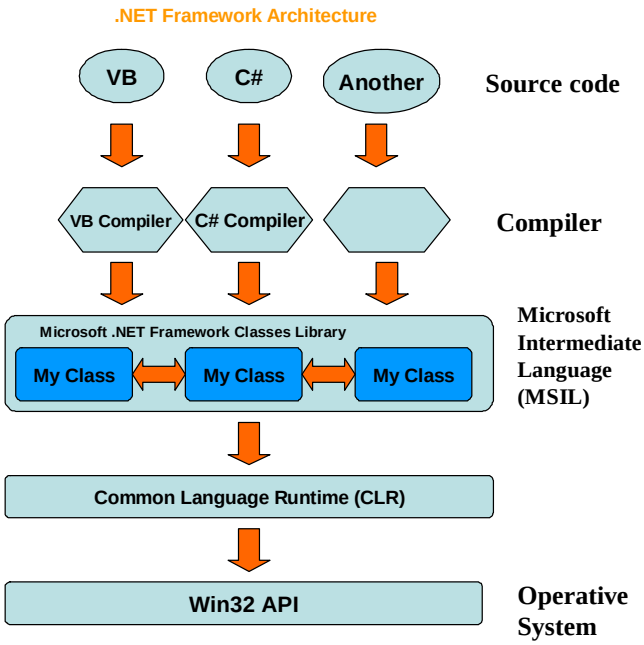


Figure 53: .NET Framework Architecture

. Net Framework supports multiple programming languages, and even though every language has its own characteristics it is possible to develop any application with any of these languages. There are more than 30 .Net adapted languages, from the more know ones (such us C#, Visual Basic or C++) until others like Perl or COBOL.

The kernel of the .Net Framework is the CLR: the execution environment where the applications developed in the different languages are loaded, extending the services set offered by the standard operative system Win32. The development tool used compiles the source code of any language into the same code, named intermediate code (MSIL, Microsoft Intermediate Language). The generated code is always the same for every language. This code is transparent to the application development, since the compiler automatically generates it.

In a second step, the JIT (Just-In-Time) compiler generates the real machine code that is executed in the platform that the computer owns. In this way some independence from the platform can be achieved.

When an application is being programmed, many times it is needed to carry out actions such as file manipulation, data access, system-status knowledge, security implementation, etc. The framework organizes all the system functionality in a hierarchical name space so when programming it is quite easy to find what it is needed. For it, the Framework owns a universal type system, called Common Type System (CTS). This system allows a programmer interact the types included in the Framework (.Net classes library) with the ones created by himself. In this way the user makes the most of the advantages of the object oriented programming, such as predefined class inheritance for the creation of new classes, or the polymorphism to modify or amply existing behaviours.

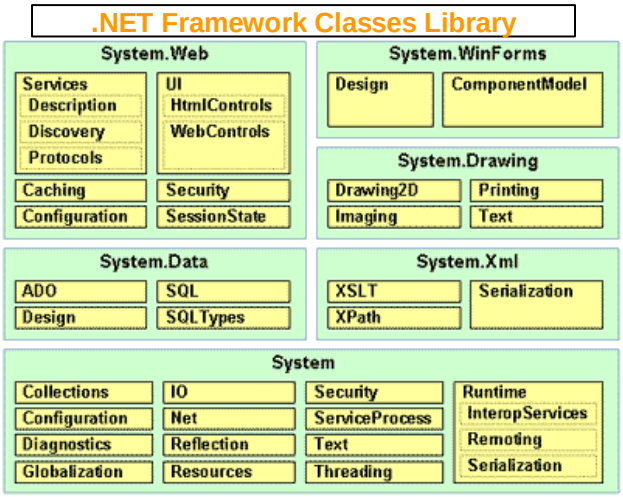


Figure 54: .NET Framework Classes Library

The .Net Framework classes Library includes, between other, three key components:

- ASP.NET [17] for the construction of applications and Web Services.
- Windows Forms for user interface developing.
- ADO.NET for database applications connection.

6.1.2 Visual Studio .NET

Visual Studio .NET is the integrated development environment (IDE) that is being released in conjunction with the .NET framework. It offers many advantages and productivity gains when developing .NET applications both for the Windows desktop and for the Web, including:

- ✓ Visual development of web pages.
- ✓ Drag & Drop web form design.
- ✓ IntelliSense and automatic code completion.

- ✓ Integrated debugging.
- ✓ Automated build and compile.
- ✓ Integration with the Visual SourceSafe control program.
- ✓ Fully integrated, dynamic help.
- ✓ It gives support for different languages (Jscript .NET, Visual Basic .NET, C#, ASP.NET…) sharing the same general structure for all of them, so the user doesn't have to learn a different environment for each different language.

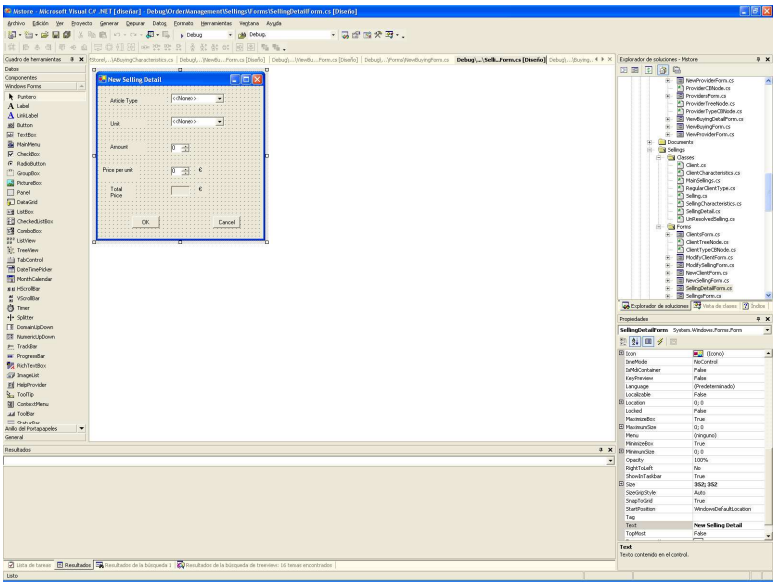


Figure 55: Visual Studio .NET appearance

6.1.3 Some advantages of the .Net platform

Here there are included some of the more important advantages provided by .Net Framework:

- **Managed Code**: The CLR makes an automatic code control so that is safe (the application is executed correctly).
- **Multilanguage interoperability**: The code can be written in any .Net compatible language as it is always compiled in intermediate code (MSIL).
- **Just-in-Time compilation:** The JIT compiler grows the application performance, as the code is specific for each platform.
- **Garbage Collector:** Automatic system for the memory management. The CLR detects when the program stops using memory and it frees the memory automatically.
- **Code Access Security:** It is possible to apply different security levels to the code.

Process such as garbage collection or the management of the code introduce overload factors that affect in the demand of more system requisites. The managed code provides a faster development speed and more security. The bad point is that the resources consume is much bigger, but nowadays, with the current processors this is not a big disadvantage.

The code administration level depends on the used language. C# allows the code administration code in a manual way, being by default an administered language.

6.1.4 Introduction to C#

C# is the new general propose language designed by Microsoft for the .NET platform. Although it is possible to write code for the .NET platform in many other languages, C# is the only one designed specifically to be used there, so programming in C# is easier and more intuitive than in other languages because it lacks the inherited elements not necessary in .NET. By this reason it is usually said that C# is the native .NET language.

The syntax and structure of C# [13][24] is quite similar to the one used in C++, because it was tried to make easy the migration between code written in other languages and C# ad well as make easy the learning process to the developers. However, the simplicity and the high productivity level are similar to the ones of Visual Basic.

Java could be an ideal language for these purposes, but due to problems with the company that created the language (Sun) Microsoft has developed a new language adding to the language some modifications to make it better.

Even while C# is relatively new, the language has been quite used, as Microsoft wrote most of the BCL using it, so the compiler is the most debugged and optimised between all the present in the .Net Framework SDK.

6.1.5 C# - Java comparison

The previous experience of the MSTORE programmers (1 year) using the programming platform Java/J2EE allows us to make a small analysis comparing platforms, as well as the languages C# and Java. Using our prior experience and some objective papers [4] [8] [26], the weak and the strong points of each language have been written down and a small comparison between them have been made. On the contrary as with the .NET platform, there is not going to be made an overall description of the J2EE standard. It is assumed some familiarity with this language, for more information consult the references.

- The only language supported by J2EE is Java (excluding that it can be accessed with JNI and CORBA). In the other hand, Microsoft .NET offers official support for Visual Basic .NET, C++. NET, C#, COBOL, Delphi, etc. … It is also offering interoperability between all these languages, so a programmer can build one component in one language and introduce in an application written in a different one. This is a positive feature since it allows an easy migration for the old programmers. It is also negative because the existence of various programming languages in a company can make the productivity decrease.
- Several companies offer development environments for the J2EE: Forte (Sun), Visual Age for Java (IBM), JBuilder (Borland), and many more. Even while they are all offering very good products, it is difficult to achieve the use facility and integration level (Multilanguage code editor, compiler, resources editor, database connection, XML editor, online help…that are found in Visual Studio .NET. The Java2EE IDEs are also written in Java, so they need more resources that the same application used for any concrete operative system.
- Both J2EE and .NET offer ability to achieve rapid application development that are not equal but are comparable. The differences are minor and it is difficult to make a compelling argument either way.
- J2EE is a more stable platform than J2EE. Some months after the appearance of the .NET Framework, there is a Service Pack available to correct the bugs of the first

version, fact that can give little confidence to the developers when using the product. J2EE was in the market 3 years before than .NET, in this time J2EE platform has been used in many projects so it has been improved and debugged. In the other hand, .NET is a more modern platform so it has included new technologies not included in J2EE.

- It is difficult to compare the performance of both platforms, as it looks like that there are not many reliable objective results that are valid. There is document published by Microsoft about the implementation in .NET of a Java application. As it is a Microsoft document it is clear that is favourable to his platform, but there are also included some points where it is explained why the document should be considered as valid. Anyway the general opinion is that it can be considered that the performance obtained by .Net platform is better than the one obtained with J2EE.
- Even while J2EE is supposed to be a standard and not a product itself, Java implementations are not 100% compatible as different vendors can add different features that the other competitors. Anyway, all the companies that offer products based in J2EE have versions for different operative systems, while Microsoft .NET only works with Win32-based platforms. With .NET the user is losing portability, but for Windows solutions it is supposed to have a better performance, and it ensures a bigger integration.

Even while the 2 platforms are independent, it is thought that they can converge in the future. Halcyon Software has market a C# to JVM that can be included in Visual Studio .NET, and Remesoft has launched Java .NET, a tool to execute native java code in the .NET environment.

Concluding, it can be written down that two excellent platforms for software development, e-business and web services were compared. It is quite difficult to decide whether a platform is better for a concrete application, as the different reflections made before have to be considered.

J2EE offers architecture with advantages such as portability, maturity and support from different companies. It also needs bigger human and economic resources. It is possible that the future users of this platform will be big corporations that maintain very heterogeneous systems, such as public administrations.

On the other hand, Microsoft .NET offers a cheaper solution with bigger performance, scalability and easier to introduce than J2EE solutions. As bad points, it only has support from one company (even if it is Microsoft) and it needs more maturity in the market. Its potential users are small and medium companies.

6.2 NHibernate

When using an application, it is usually necessary to store the information it is being used among different executions of the application. If it is translated to an object-oriented developer view, it is needed some way to store the objects that contain useful information so that they can be used in other executions of the program. This is what is usually called persistence.

There are several ways to achieve this goal. For instance, Java includes Java Beans Enterprise, that allows making the objects persistent in a transparent way. The objects are saved into files and the next time it is wanted to use them it is an easy task to recover them. However, the user looses the benefits of working with a relational database.

. NET includes ADO.NET for database connection. Anyway working with object-oriented software and a relational database can be a thorny task, as the user has to write SQL sentences to get the objects in and out from the database. In order to save development time it is a good idea the use of any tool that makes this work for the programmer.

NHibernate [11] [20] is a .NET based object persistence library for relational databases. It is based on the Java Hibernate relational persistence tool. It allows the persistence of the .NET objects from an underlying relational database. In another words, it achieves the mapping of a data representation from an object model to a relational data model with an SQL-based schema. Instead of generating SQL sentences, the code is only concerned with my objects. NHibernate generates the SQL sentences and locates the information in the different tables and columns (provider transparent persistence).

The library is in the beta stage, so it does not warranty that it is free of bugs (and as the researchers have checked, the existing documentation is much poorer than the Java Hibernate´s one). Anyway, after the basics are understood, it helps a lot the task of translating the objects into tables in a relational database.

Nowadays, the translation between a .NET class and a table has to be done manually. In a future, NHibernate is going to include tools that help in generating a schema and generate classes from mapping files. Basically, there are 5 steps a user has to follow to make our objects persistent:

Step 1: Create the table where the information is going to be stored.

Step 2: Create the class is wanted to be made persistent.

Step 3: Create a configuration file that allows NHibernate connect to my database.

Step 4: Create a mapping file where is telling NHibernate how the class properties should be translated to the database table.

Step 5: Use the NHibernate API to make the class persistent. These API has methods to initiate the connection, connect the database, get/save objects from/in the database, update the tables, disconnect the database, and more operations.

The researchers' experience with NHibernate was not very positive, as it was quite hard to find the propped documentation to solve the problems they had. Anyway, as it is based on the Java Hibernate tool (that warranties a good performance) it is expected that these problems will be solved in later versions and the documentation will be improved.

6.3 EA

Developing an Object-oriented application is much more than developing a class model - lifecycle of system development - business process analysis, use case requirements, dynamic models, component and deployment, system management, non-functional requirements, user interface design, testing, maintenance etc.

For this reason there should be chosen the appropriate CASE tool for improving the design of our application.

6.3.1 UML

The Unified Modelling Language [16] has quickly become the de-facto standard for building Object-Oriented software.

The OMG specification states:

"The Unified Modelling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

The important point to remark here is that UML is a “language” for specifying and not a method or procedure. The UML is used to define a software system; to detail the artefacts in the system, to document and construct. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process) - but in itself it does not specify that methodology or process.

6.3.2 Features within Enterprise Architect UML Modelling tool

The chosen tool was Enterprise Architecture [22], and the following characteristics are the main reasons for this election.

- ✓ Comprehensive support for UML 2.0
- ✓ Comprehensive and flexible documentation
- ✓ Forward and Reverse Code Engineering for C# (used development language), and also for other languages like Java or VB
- ✓ Easy to use
- ✓ Support for testing and maintenance.
- ✓ XML import and export
- ✓ Spell Checker.
- ✓ Support for Glossary.
- ✓ Reasonably price.

Although it has not a lot of extras (most of them useless) as Rational Rose or Together, the services that EA provides are enough and its low cost and great functionalities, are the reasons of the choice.

6.4 NUnit

Remembering the objectives of this project, in this first version it is going to be implemented only a part of the framework since:

- It would be a hard work (almost impossible) to implement a useful, complete and efficient store management Framework in only one interaction and in the given time.
- It is better to start developing a part and then make that part bigger and bigger in several versions.

So, one of the destinations of the code done in this version is to be reused in other programs.

For do this successfully, it is necessary to document the Framework, and also to **test** all the modules independently, for minimizing the probability of errors when they work together, trying to obtain a bug-free and quality code.

It is important to point that the part of the program that should be tested deeper is the one that is included in the Framework (implementation of the methods of the classes that are really going to be reused). It is not so important to test the interface classes, since these example applications are not going to be used by real customers. Their objective is to demonstrate some characteristics and advantages of the Framework.

There is some testing software that makes this task easier. A unit test is nothing more than the code wrapper around the application code that permits test tools to execute them for fail-pass conditions.

In this version it was used NUnit because [21]:

- ✓ NUnit is a unit-testing framework for all .Net languages, so it is appropriate for C#.
- ✓ Initially ported from JUnit, the current release, version 2.2, is the fourth major release of this xUnit based unit testing tool for Microsoft .NET.
- ✓ It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities. NUnit brings xUnit to all .NET languages.
- ✓ It is open source
- ✓ It uses an Attribute based programming model.
- ✓ It loads test assemblies in separate application domain hence an application can be tested without restarting the NUnit test tools. The NUnit further watches a file/assembly change events and reload it as soon as they are changed. With these features in hand a developer can perform develop and test cycles sides by side

6.5 Web Services

Web services [17] are very similar to web pages. Normal web pages allow interaction between the client browser and the web server that hosts the web page. However web services are used strictly for one program that interacts with another and has no user interface.

They should be independent of the operating system and the programming language used on either the server or the client side. The only requirement is that both server and client support the standard protocols HTTP (protocol used by the Web), SOAP (cross-platform standard for formatting and organization information), & XML.

Web services allow that one object located on the server expose program logic to clients over the Internet. In short, a web service is a function or a method call over the Internet. The web services infrastructure has several defining characteristics:

- Both the web service server and the client application are connected to the Internet and are able to communicate.
- The data format with which the two ends communicate conforms to the same open standard (usually the SOAP protocol).
- The systems at the two ends of the connection are loosely coupled.

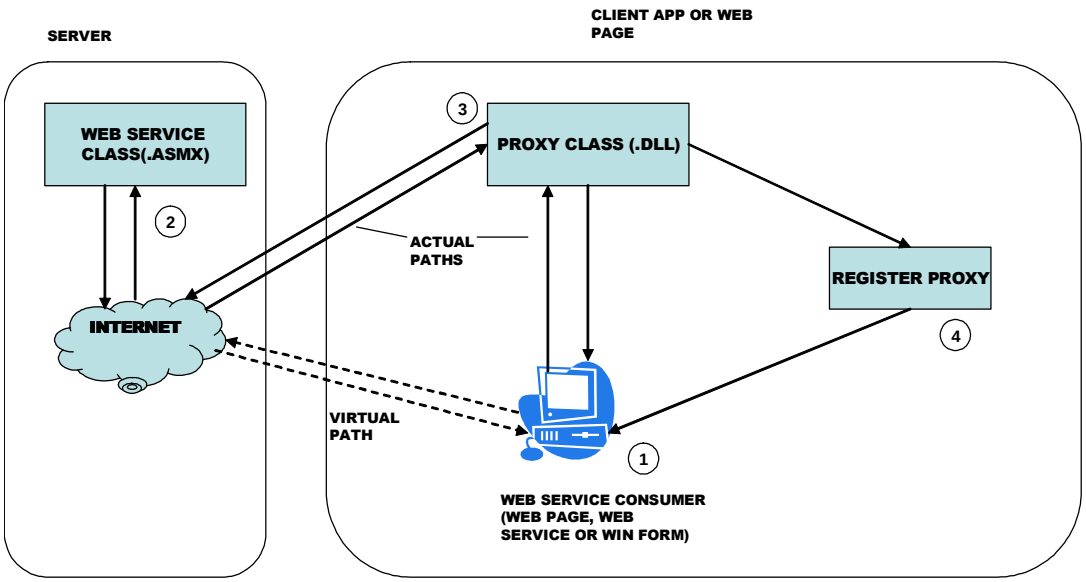


Figure 56: Logic behind a Web Service

This figure shows the logic behind the web services process. A web service consumer (for instance a program, number 1) makes a call to the web service (number 2). The consumer thinks it is talking directly to the web service over the Internet, but it is false. In fact, the actual call is being made to a proxy class (number 3), which is local to the consumer. This proxy handles the complex infrastructure of sending the request over the Internet to the server machine and getting results back. All of this is made because the proxy was previously registered with the consuming application (number 4).

Chapter 7. Conclusions

This document tries to make a complete explanation about the reasons that motivated the creation of MStore Framework, the process of designing and developing its components and structure and finally the real application of it, trying to build some particular examples, with a specific problematic each one.

The project was done with a quite important academic motivation, so in the beginnings there were chosen a wide variety of technologies useful to complete the skills acquired during the academic life of the authors and preparing them for future job: .NET platform and C# (in comparison with Java), web services –rising nowadays-, experimenting the advantages of design patterns and frameworks, and going in depth in management applications, largely demanded in companies.

Although the project started as a preliminary study, future versions will use the work for developing more usable real-world applications. The size of the project provides work for at least three generations of students continuing it, as was thought from the beginning. Guidelines for the future work will be given later.

Anyway there are several conclusions that have been taken from the process, although most of them can be found along the report. Here there are exposed the main ones.

- ✓ **Patterns** are a good way to describe frameworks because first-time users of a framework will usually not want to know exactly how it works, but will only be interested in solving a particular problem. The benefits of using patterns and 3-layer design are explained before, but in the scope of this project it is difficult to make this affirmation, since the creators of the framework are the first (and only, by now) users of it. It is expected that future users will enjoy the benefits of the well-structured design, and that most of the adopted solutions will be used in future versions.
- ✓ The particular applications developed using MStore have seen its **development time** decreased, since most of the code used in the business logic there was already written in the framework. The design, development and debugging of the framework took four months of work, and the construction of the particular applications took one month and a half. Make predictions about development time reduction using MStore is quite difficult, as some considerations must be taken into account.
    - o When a developer uses a framework for first time, it is assumed that he is going to spend an important percentage of his time understanding its operations and thinking how to apply it to the particular problem he is trying to solve. This fact cannot be contemplated this time, since the developers had previous knowledge about the framework design.

      For decreasing the **learning time**, the documentation plays an important role. A good, clear and standard version of the Framework documentation can

hugely help to the future user of MStore when trying to understand or expand it.

- The applications built using the framework have all the benefits of the design patters, so they are easy to read and to maintain. Developing one application with these characteristics from scratch can make much bigger the development time.

Taking into account these considerations it can be concluded that development time using the framework will be at least half of the normal development time, assuming that the program has a medium difficulty. If the rate of difficulty increases, the relative time comparing to the one spending without the framework decreases.

In our particular study case, the development time for each of the applications was two months (including analysis phase), but as this time is including some debug from the framework, it should be concluded that the development was done in 4 weeks. The development time if MSTORE had not been used wouldn't be smaller than two months.

- ✓ On the other hand choosing the **technology** is also an important decision. Working in a comfortable, easy to use, useful and familiar environment helps the development and reduces the programming time. Also Visual Studio makes much easier the realization of the web services, providing wizards and methods that do the task almost automatically. Related with the language C#, it was easy for the programmers to learn it since it is quite similar to the previously used Java. Knowledge of both platforms gives the programmer a broad vision of the current market.

- ✓ Although the authors have worked in group before, for them this is the first big application **developed in pairs**. In early stages of the project it was planned to use a methodology such as Extreme Programming, but finally it was not possible and it was decided to separate the project in two parts, being in charge one of each part. From the experience it can be concluded that work in a group (in a couple this time) can be a hard task, and it is necessary a lot of time organizing the future work and talking with the colleague. Following any methodology could help in this task, but it is fundamental that the programmers are in a good attitude to the partner.

7.1 Future plans

By now, it cannot be said yet that MSTORE Framework is a robust and probed support for creating new management applications. A program with this extension needs further research and debugging until it can be used by any end-user. Next versions of MSTORE could add the next features:

- ✓ Repair possible bugs that can appear in this version. Expand the web services offered and develop consumer programs that explode all its possibilities.
- ✓ Probe the rest of the Framework programming applications that use as many as possible number of functions. It is expected that these applications will have a medium size, as the framework is not thought to be used by big department stores. Expand the web services offered and develop consumer programs that explode all its possibilities.

- ✓ Integration with other parallel programs that are been developed now, such as risk management or document management.
- ✓ Thinking in other programs that can complement MSTORE as an accounting one. Publish the program and try it with real users, pondering strong and weak points.

References

[1] Apple, J.M. (1991). ***Plant Layout and Material Handling.*** Krieger Publishing Company.

[2] Aquilano, N. J. & Chase, R.B. (1991): ***Fundamentals of Operations Management.*** Irwin.

[3] Beck, K. & Johnson, R. ***Patterns generate architectures.*** University of Illinois

[4] Chamorro, R. (2002). ***Análisis comparativo entre Microsoft® .NET y Sun® J2EE.*** <http://www.ciberteca.net/articulos/programacion/net/>

[5] De la Fuente, D. ***Organización de la producción para Ingenieros*** [Production organization for engineers]**.** University of Oviedo

[6] Deusch, L. P. (1989) ***Design Reuse and Frameworks in the Smalltalk-80 Programming System***. ACM Press.

[7] Discrito K S.L. (2003). ***IOne Pyme***

[8] Farley, J. (2000). ***Microsoft .NET vs. J2EE: How Do They Stack Up?*** <http://java.oreilly.com/news/farley_0800.html>

[9] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. ((1994)**. *Design Patterns Elements of Reusable Object-Oriented Software***. Addison Wesley Professional Computing Series.
And in the Web: http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/

[10] Gamma, E. & Eggenschwiler, T. **JHotDraw Web Site** [online]. <http://www.jhotdraw.org> [Query: 1st October 2004]

[11] Gehtland, J.(2004) ***Nhibernate.*** <http://www.theserverside.net/articles/showarticle.tss?id=NHibernate>

[12] González Pisano, J. & Rodríguez, M. **Framework MSTORE Web Site.** <http://www.student.luth.se/~javgon-4/>

[13]Gunnerson, E. (2000). ***A programmer's introduction to C#.*** Apress.

[14] Johnson, Ralph E. (1992). ***Documenting Frameworks using Patterns***. Champaign: University of Illinois

[15] Lago, C. ***Univesity of Oviedo*** <http://www.uniovi.es>

[16] Larman, C. (1997)**. *Applying UML and Patterns***, New Jersey: PTR.

[17] Liberty, J. & Hurwitz, D. (2002). ***Programming ASP.NET .*** O´Reilly

[18] Microsoft. ***.NET Framework Home.*** < http://msdn.microsoft.com/netframework/>

[19] Microsoft . ***.NET Framework Download.*** <http://msdn.microsoft.com/netframework/howtoget/default.aspx>

[20] Morris, J. Et al. ***Nhibernate project.*** <http://nhibernate.sourceforge.net/>

[21] Newkirk, J. W., Two, M. C., Vorontsov, A., Craig, P.A., Poole, C. ***Nunit Home Page.*** <http://www.nunit.org/contactUs.html>

[22] Newman Creative. ***Enterprise Architecture WebSite.*** <http://www.sparxsystems.com.au> [Query: 16th October 2004]

[23] Plossl, G. (1985). ***Production and Inventory Control***. Boston: Prentice-Hall

[24] Schmidt, D. C. (1997) ***OO Application Frameworks.***
< http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html>

[25] Robinson, S., Nagel, C., Skinner, M., Evgen, B., Watson, K. & Glynn, J. (2001). ***Professional C#.*** Wrox Press.

[26] Vawter, C. & Roman, E. (2001). ***J2EE vs. Microsoft .NET***. ***The middleware company.*** <http://www.theserverside.com/resources/article.jsp?l=J2EE-vs-DOTNET>

Appendices

I. Storage systems and product location in a Store

In first view, it can be differentiated two different types of storage: open air and covered.

From the point of view of stored merchandise type and equipment for its manipulation, the storage systems can be grouped in function of three different criterions [1][5]:

1. According with the organization for locating the merchandises.

   a. Tidy Storage

   We assign to each product, one only place, fixed and predetermined. Therefore, the spaces, destined to accommodate the different products, adapt themselves to the particular characteristics of those products, and they only accommodate products with those characteristics.

Emphasize in a positive form in this system, the facility of control and manipulation of the products. However, the limit in the storage by the foresee spaces can cause under utilise of it, since it is possible that the capacity does not be covered in whole.

   b. Chaotic Storage or free space

In this storage system, we assign spaces when the products or merchandises are tidied up, without keeping any predetermined order.

Nevertheless it is common to establish some rules for looking for the location of the products (security reasons, optimisation of routes, environmental conditions, etc.) and the dimensions of the spaces will be the appropriate for the products that they can receive.

This method allows a better use of space but it is required sophistic control methods.

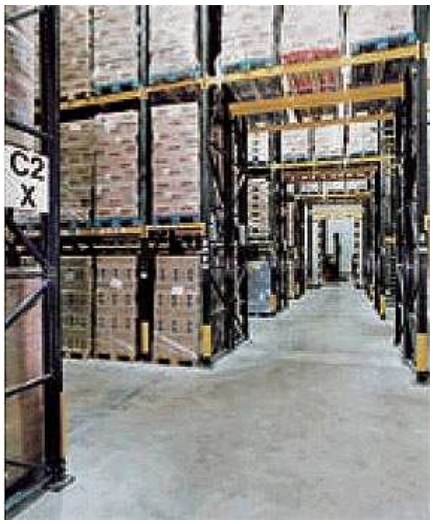


Figure 57: Chaotic Storage

2. According with input/output flux.

   a. LIFO Storage

The last product that goes into is the first that goes out.

Figure 58: Dynamic by gravity-FIFO

b. FIFO Storage

The first product that goes into the store area is the first that goes out. It is the most appropriate for perishable products or fast caducity.

3. <u>According with the equipment used for optimising the available space.</u>

a. Without corridors Storage

Between the products there is not any space.

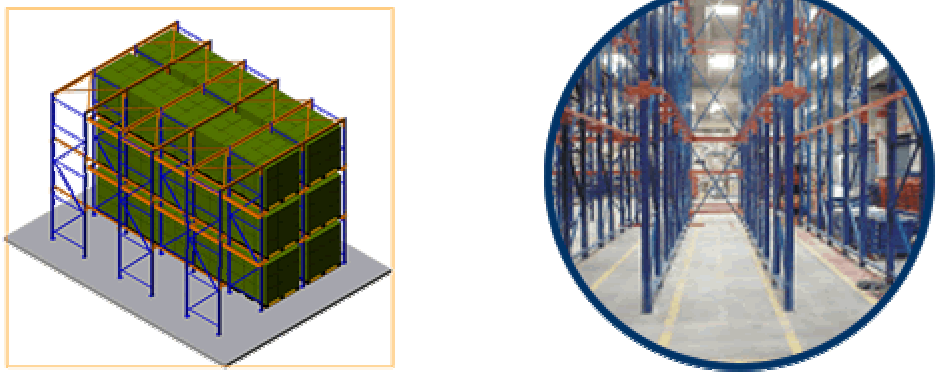


Figure 59: Rack

b. With corridor Storage

Regarding the merchandise location systems, it can be pointed out that it is very important to be able to localize in all the moments any product.

Location is the space where the product is accommodated temporally. Each location has to be identifying by a code.

Letters or numbers composes this code, and it allows identifying: the location of the merchandise in the shelves, corresponding zone and store used.

There are two main Location Systems:

1. <u>Shelves location or lineal location</u>

A correlative number has to be assigned to each shelf. Correlatives numbers identifies the deep of that shelf too, with origin in the headboard. The level can be identified with number from the low level to the high level.

2. <u>Corridor location or comb location</u>

Each corridor is identified with a correlative number. Each corridor is only route in one-way, alternating ascendant way with descendant way. The deep of each shelf is numbered in ascendant way, assigning even numbers to the right and odds numbers to the left, starting the numeration of the following corridor in the other extreme.

So it can be defined with three coordinates any location in the store. The code can use the form: A, B, C, D where:

A: zone in the store

B: shelve or corridor

C: deep

D: level in shelves.

II. Stock Management Policies

Why inventories

Almost all the organizations own stocks of many types, useful for making operations in the store, invest money, deterioration, etc, and there are some costs attached to the stocks. One obvious question is why do organizations use stock? In a first approximation, it serves two purposes. First, it allows the company to manufacture goods away from the actual customer. Second, it allows management to separate production and distribution processes.

There are much more reasons for the stock possession, we can stress as the usual ones [23]:

- Smaller prices in big orders
- Buy articles when the price is small and it is expected that it increase.
- Buy articles that are no more produced or that are difficult to find.
- Separate stages in production process.
- Providing solutions for emergencies.

Regardless of the reason for having stocks, there are some costs related to them and they are usually surprising highs. One of these costs is the annual possession stock cost, which can vary from 15% of the article buy value to the 50%, with a medium of 25%.

As the associated costs are very high, there are obvious incentives to look for policies that reduce them. First serious study about stocks was made by F Harris and written in 1915 about operations and costs. The most known work was made by R. Wilson with his famous formula, written in a book published in 1934. Nowadays, computer systems and he Japanese production ideas have taken the lead, and more effective improvements are still producing. This fact can be seen as the inventories changed from 35% of GDP in the sixties, to 25% in the middle eighties.

Types of inventory

- **Movement inventories:** They are needed because manufacturing the products and transporting them is time consuming. In-transit inventories are those, which are kept for covering transportation time losses. Work in progress inventories are the ones used during manufacturing. Management can control these inventories by changing production/distribution system. For instance, WIP levels can be changed by altering manufacturing process, batch size or production schedules. Similarly, adopting a different mode of transport, or reducing distance between supplier and factory can affect in-transit inventory.
- **Organisation inventories:** They help to separate the manufacturing processes from distribution system by the finished goods inventory. The finished goods inventory helps cater to the demand from distribution centres directly. This relaxes the schedule of assembly (manufacturing) operations. The other types of organisation inventories are cycle stock, safety stock and anticipation stock.
- **Cycle stock:** It is the stock that arises by producing larger quantities of end products than that which is required immediately. This is done because producing a large batch of items could be cost effective.

- **Safety stock**: It provides a cushion against the deviation of demand or the supply as was forecast. This assures that customer needs are always met immediately.
- **Anticipation stock**: Such stocks are held for items which occasionally raise in demand and whose production patterns are more inflexible. These anticipation stocks are deployed during high demand periods.

Factors to consider in stock management

If it is wanted to develop some policies for the stock control, an obvious starting point it to know the costs with more detail, as well as the demand, supply time, etc. Next are described all the factors that have important influence in the inventories.

- **Acquisition cost**: This is the final price of the article given by the provider (the unitary price of the product). Usually here is included more information such us the transportation to the store, unpacking, etc.
- **Possession cost:** Assumed cost for storing a unit of an article during a given time period, the cost is proportional to the stored amount and to the time it is in the stock.
- **Emission cost:** Cost attached with making an order, it must include the total costs of the order (duties, insurances, packing, quality control…).
- **Breaking cost:** If one article is needed but it cannot be used, there is usually one cost related with this shortage. In the simple case, there can be only lost the direct winning of a sell, but usually should be also considered loosing of clients and future sells. Breaking costs are very difficult to evaluate, but they are usually quite highs.
- **Demand**: It is a fundamental variable in stock management. It should be considered is know in certainty of risk conditions, if it is uniform or suffers variations, etc… When the demand is variable, he management models are more difficult to solve
- **Supply time:** It is the time between the moments I make an order until I can use the articles. This information can be based, as before, in statistical methods.

Quantitative models used in stock management

The quantitative models are very useful in distribution environments. It is an alternative to the material management techniques (MRP, DRP), used in manufacturing process and very popular during last years. A quantitative model uses having into account: a list with the costs, the model demand, order size, etc., and it is used to find the order type that minimizes the cost.

One of the questions the stock management system tries to answer is when should be done one order. It will depend of the stock control system used, the demand, the article value, the supply time, etc.. Basically, there are two different order policies:

- "Fixed order amount", where I make a fixed-amount order when it is considered that the product is least that a given level. The variable it is tried to determine with this system is the order size, called "Q*", that is constant in each emission order.
- "Fixed period system", where the orders (with different amounts) are made in regular time intervals, in order to increase the stock in a specified value. The important variable is the "T" time, between two successive orders, being this time constant during all the management period.

The other question, how much we should put in each order, is implicit in the last question. If the orders are made with high frequency, the stocks are high and the order size is small, it the frequency is low the order size is going to be bigger.

Trying to balance these two methods, one variation is to calculate an “Economic Order Quantity”, EOQ. There are also other quantitative methods such applying discounts, variable demand, etc.